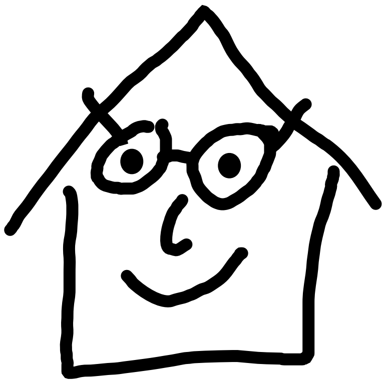


The *Home2L* Book

Clever Tools for a Private Smart Home

Gundolf Kiefer

Version: 368d*

November 28, 2018

Contents

1 Introduction

1.1 What Are the *Home2Ls*?

The *Home2L* `[houmtu:l]` suite is a framework, library and set of tools for automation in private smart homes. Its main features and design goals are:

Efficient and Lightweight Design

All core components are written in C/C++, with a very minimum set of external dependencies beyond `libc` - ideally suited for small embedded devices and microcontrollers. There is no need for a Java runtime environment or heavy web frameworks. Starting up a server and command shell and shutting them down again takes less than one second altogether - on an ARM-based minicomputer running at 144 MHz!

Ambient Intelligence, No Need for a Central Server

Central servers are single points of failure. *Home2L* follows a completely distributed concept. Any (mini-)computer can act as part of the network. If resources, such as sensors or actors, are connected to them, they can be exported to any other host in the *Home2L* network. A failure of a host only causes its own resources to be unavailable - everything else keeps on working.

Automation Rules Written in Python - But not Limited to That

There is no new language or tool to learn to formulate automation rules. *Home2L* rules are typically formulated in Python, they profit from the both the simplicity and the power of the Python language. There can be multiple rules scripts, they may run on any machine, and they may by combined with other software routines.

Other ways to interact with *Home2L* resources is via the C/C++ API from any application or by shell scripts using the *Home2L Shell* in non-interactive mode.

Easy Driver Development in C/C++, Python or Any Other Language

An API for *resource* drivers allows to easily add support for new hardware. A driver can be implemented

- in native C/C++ code (e.g. *GPIO*). - as a shell script (e.g. *Weather*). - in Python (examples can be found in the tutorial).

Privacy

The *Home2Ls* do not need any Internet connection and do not try to communicate with hosts other than they are configured to. By design, the *Home2Ls* communicate with each other over a (trusted) LAN, which can easily be set up and secured using standard Linux/UNIX techniques. The open source licensing ensures transparency for what the software does inside the user's private home.

Modularity

The core part, the *Resources* library, is kept small and portable with APIs for application programs *and* drivers in C/C++ and Python. All other components are optional and can be used or replaced by alternatives as desired by the user.

Figure 1.1 shows an overview on the *Home2L* components.

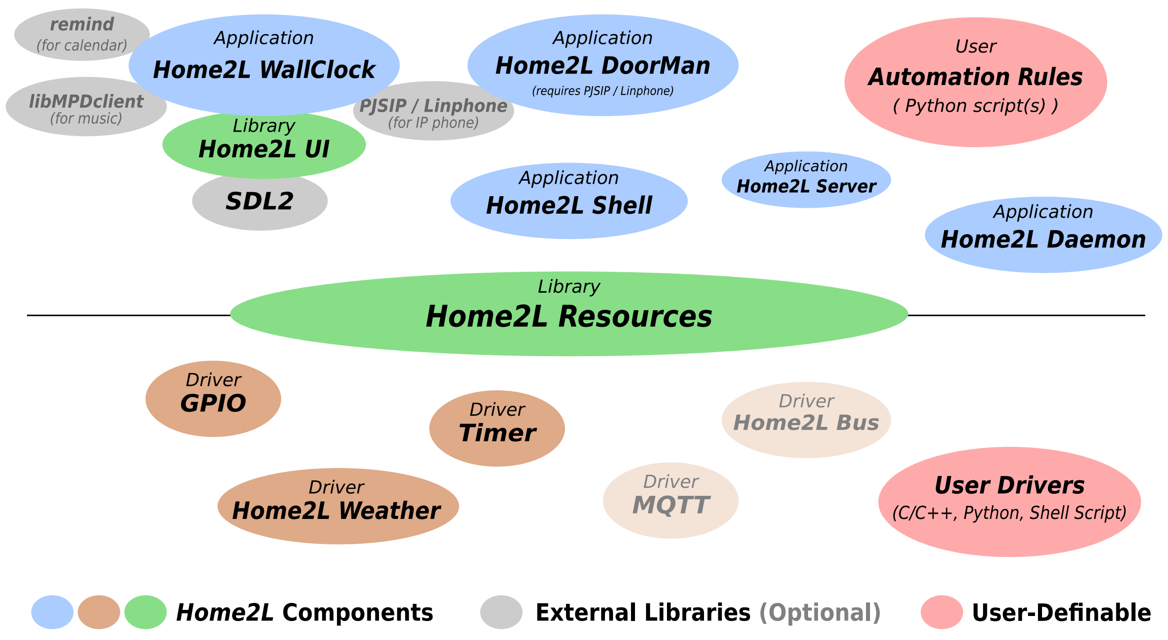


Figure 1.1: *Home2L* Components: Tools, Drivers, Libraries

1.2 About This Book

This book serves as the central user and reference manual for the *Home2L* suite.

Further information can be found in the C/C++ API and the Python API documentation. Consulting the API documentation is particularly helpful for writing own resource drivers, sophisticated automation rules or applications using some *Home2L* library.

Presently, the *Home2L Book* is still under construction and primarily targets readers with computer skills. Not all existing features are already documented in this book. Extending the book to provide good end-user documentation is subject to future work (see Section 1.3).

Chapter 2 contains a step-by-step tutorial covering the installation, concepts and all core tools of the suite. This should be the starting point for any new user.

Chapters 3 and 4 describe the installation and management of an installation, respectively. Chapter 5 explains the concepts and usage of the *Resources* library, the core component of the *Home2L* suite. This is followed by instructions on writing automation rules (Chapter 6) and drivers (Chapter 7). Chapter 8 documents the drivers contained in the *Home2L* base distribution. The remaining chapters cover the main user applications, presently the *Home2L WallClock* (Chapter 9) and the *Home2L DoorMan* (Chapter 10).

1.3 How Can I Help?

Until now, the *Home2L* project has been developed by a single private person in his spare time. The code has been published with the hope that is useful to the community.

To let the project grow further and make available to a wider audience, volunteers are needed and welcome.

Great contributions for the community would be, for example:

1. Make **sample installations** and document them.
2. **Packaging**: Create packages for major Linux distributions.
3. **Documentation**: Write good documentation, particularly for end users.
4. **Drivers:** Implement drivers for any hardware you like or have. Help with an *MQTT* bridge would be appreciated, too.
5. **Bridges to other home automation frameworks:** Implement drivers to interface with other open home automation frameworks.
6. **Report and help fixing bugs.**
7. **Translations** and internationalization.
8. Write an **HTML frontend**.

For any questions on how to participate, do not hesitate to contact the author via the project page.

2 Tutorial: Have a Test Drive!

2.1 Introduction

2.1.1 Overview

This step-by-step tutorial aims to give an exhaustive introduction to the *Home2L* suite covering the installation and all core tools and concepts of the suite. It should be the starting point for anybody new to the software.

During the tutorial, you will

- compile and install the *Home2L* suite.
- learn about the *Home2L* concepts: distributed and modular design, *Home2L Resources*, tools, drivers, automation rules.
- learn about *Home2L Resources*: values and states, subscriptions, requests, the directory.
- use the *Home2L Shell* for inspection, maintenance, and logging.
- get acquainted with several tools: `home2l-shell`, `home2l-wallclock`, `home2l-daemon`, `home2l-server`.
- explore the *WallClock* (including the music player and calendar).
- learn how to write automation rules.
- obtain and understand a sample installation, which can be the starting point for your own home installation.
- get references to further information.

The tutorial is carried out on a virtual machine (VM) running Debian Linux, which will be created as the first step. This just takes very few minutes and makes the *Home2L* installation process fully transparent, since no pre-configured boot image is necessary.

Sections 2.2 - 2.4 deal with the installation. Section 2.5 will guide you through the *ShowHouse*, a virtual example *Home2L* installation. Section 2.6 introduces the Nhome2L Shell together with the basic concepts about Home2L resources. The remaining subsections cover dedicated topics and can be exercised independent from each other, depending on your interests and preferences.

2.1.2 Notational Conventions

The tutorial uses the following conventions in notation:

a) A black triangle (▶) marks an instruction to follow.

b) `Typewriter text with a grey background` marks an interaction with your computer. Lines starting with a prompt indicate commands to be entered into the respective interpreter:

   `$` – the Linux shell (`bash`),

   `home2l>` – the *Home2L Shell*,

   `»>` –the Python command interpreter.

c) Blocks with a big italic '***i***' provide additional information.

Example:

▶ Enter the following commands to verify your computer's calculation skills:

```
$ python3
>>> 2+3
5
```

Then push *Ctrl-D* to quit the Python shell again.

i This is some additional information, not essential to just run the tutorial successfully.

2.2 Setting up a Virtual Machine

i If you already have a machine (physical or virtual) running Debian Linux or some derivative (other distributions should work as well, but has not been tested), you can skip the VM creation and use your existing machine – at your choice.

2.2.1 System Requirements

The following instructions have been tested with *VirtualBox 5.2.10* and a *Debian 9.6 (Stretch)* guest system.

Up to 16 GB of disk space and 512 MB RAM will be needed for the virtual machine.

2.2.2 Creating the VM and Preparing the Installation Medium

i The following steps mention Linux/Unix shell commands for illustration purposes. Of course, any OS capable of running *VirtualBox* can be used. Please use the respective (graphical) tools of your OS to download, unpack and run the virtual machine.

▶ Create an empty working directory 'home2l-tutorial' and change into it:

```
$ mkdir home2l-tutorial
$ cd home2l-tutorial
```

▶ Download and unpack the virtual machine:

```
TBD: Check
$ wget https://github.com/gkiefer/home2l/raw/master/home2l-showcase.tar.gz
    # or copy that file from a <home2l source>/home2l-showcase.tar.gz
$ tar xzf home2l-showcase.tar.gz
```

▶ Download and provide a Debian installation image as 'install.iso' in the same directory:

```
$ wget https://cdimage.debian.org/debian-cd/current/i386/iso-cd/debian-9.6.0-i386-↩
    ↪netinst.iso
    # Adapt the path as necessary. We need an i386 "netinst" image.
    # More information can be found on the Debian download page at
    # https://cdimage.debian.org.
$ ln -s debian-*.iso install.iso
    # or rename the file (if your OS does not support symbolic links)
```

The virtual machine comes with an empty harddisk image and is configured to have the CD/DVD image `'install.iso'` inserted in its optical drive.

▶ Start *VirtualBox*, select "Machine → Add..." and navigate to

```
home2l-showcase/home2l-showcase.vbox
```

to add the *Home2L ShowCase VM* as new virtual machine.

2.2.3 Installing Debian Linux

Estimated Time: 45 Minutes (mostly unattended)

▶ Start the virtual machine – either using the graphical UI of $VirtualBox$ or by running the following command on the command line:

```
$ virtualbox --startvm home2l-showcase
```

The VM automatically boots from the Debian installation medium and runs the installer.

▶ During the installation, *accept all default settings or leave fields empty*, except for the following options:

1. Select your country, language and keyboard layout as convenient for you.
2. As a computer name, enter: `home2l-showcase`.
3. Leave the root password empty (will allow `home2l` to use `sudo` for root access).
4. As the name for the first normal user enter: `home2l`
5. For user '`home2l`' enter a password at your choice (and do not forget it!).
6. In the hard disk partitioning dialog you can accept all defaults. Only in the last step, you need to explicitly select "yes" to write the partition table to disk.

   The installer now requires approx. 5 minutes without interaction to install the base system – time for a small coffee.
7. In the software selection dialog (aka "`tasksel`"), select "`Xfce`" and the `standard system utilities` (the latter may appear translated to your language) and *uncheck everything else.*

   The installer now requires approx. 30 minutes without interaction (depending on your internet bandwidth and hard disk) – time for a large coffee.
8. Install the GRUB bootloader to `/dev/sda`.

▶ Finally, confirm to reboot the system.

The installation medium is "ejected" automatically, the system reboots and you can log in as user `home2l` and use the system.

Your done!

i The VM has been pre-configured to use "NAT" networking. This is the most fail-safe setting and allows the VM to share the internet connection with your host. If you want to contact the VM from your host or some other machine in your LAN, you may change the network setting to "bridged". Please consult the *VirtualBox* documentation for details.

2.2.4 Finalizing the Virtual Machine

▶ Log in as user `home2l` and start a terminal window.

▶ Download this book into the virtual machine and open it there. This will allow you to copy and paste terminal commands from this document into the terminal by marking with the left mouse button and pasting with the middle button.

```
TBD: Check
$ wget https://github.com/gkiefer/home2l/raw/master/home2l-book.pdf
$ evince home2l-book.pdf
```

Navigate to the tutorial section.

In order to optimize the guest display, the *Virtual Box Guest Additions* should be installed.

- Add the *Debian backports* repository to `etc/apt/sources.list`:

```
$ sudo cat << EOF >> /etc/apt/sources.list

# Debian backports
deb http://ftp.debian.org/debian stretch-backports main contrib non-free
EOF

$ sudo apt update
```

- Then install the guest additions:

```
$ sudo apt install linux-headers-686 virtualbox-guest-dkms virtualbox-guest-utils
```

- For the following steps, you should maximize your *VirtualBox* window. For optimum experience, the guest display should have at least 1600x900 pixels. Make sure that the *"Auto-resize Guest Display"* option in *VirtualBox*, then enter

```
$ xrandr --output VGA-1 --preferred
```

  to activate the new resolution.

2.3 Installing Prerequisites

- Log in as user `home2l` and start a terminal window.
- Install pending security or other updates:

```
$ sudo apt purge pulseaudio   # pulseaudio has no use here and may cause problems
$ sudo apt update
$ sudo apt upgrade
```

- Install all required packages for building and running the tutorial:

```
$ sudo apt install git g++ \
    python3 python3-dev swig libreadline-dev curl \
    libsdl2-dev libsdl2-ttf-dev libmpdclient-dev gettext imagemagick inkscape \
    net-tools remind patch mpd mpc
```

What are all these packages required for?

`git g++`
: is all you need for downloading (`git`) and a minimum configuration (`make CFG=minimal`) – a C++ compiler together with the standard C library and `make(1)`.

`python3 python3-dev swig libreadline-dev`
: are required for the Python API and additional comfort with the *Home2L Shell* command line.

`curl`
: is required for the *Weather* driver (`home2l-drv-weather`).

`libsdl2-dev libsdl2-ttf-dev libmpdclient-dev gettext imagemagick inkscape`
: are the build requirements the *WallClock* (`home2l-wallclock`) in the demo configuration (with calendar and music applets).

`net-tools`
: are used by the *WallClock* info screen (and useful for system maintenance anyway).

`remind patch mpd mpc`
: are run-time requirements for the *WallClock* calendar and music applets.

- Disable `mpd` as a system service (will later be started by the *Home2L* daemon):

```
$ systemctl disable mpd
```

2.4 Building and Installing the *Home2L* Suite

- Get the source code:

```
$ git clone  https://github.com/gkiefer/home2l.git   # TBD
$ cd home2l
```

- Compile:

```
$ make CFG=demo
```

- Install and setup:

```
$ sudo make CFG=demo install
$ sudo /opt/home2l/bin/home2l-install -i
```

- Copy some tutorial files to the *Home2L* 'var' directory:

```
$ cp -a doc/tutorial/var/* /var/opt/home2l/
```

- Open a file browser, navigate to `/opt/home2l/share/home2l.desktop` and drag that file to the launcher panel at the bottom of the desktop. This will create a launcher button to quickly start the graphical *WallClock* application (see Figure 2.1).

2.5 First Steps into the *ShowHouse*

▶ Click on the *Home2L* launcher button to start an instance of the *WallClock*.

The *WallClock* (home2l-wallclock) is the main end user interface – a universal information display to be mounted on the wall of rooms or installed on mobile devices. We will explore its capabilities later.

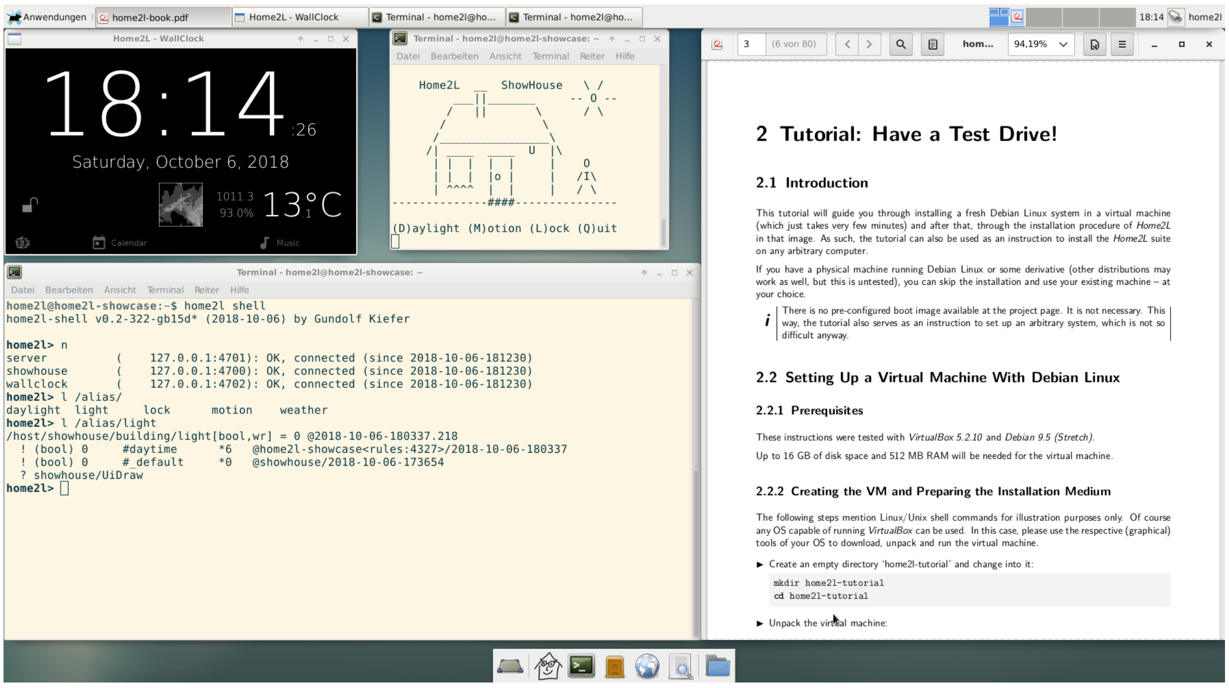
i The *WallClock* window can be resized arbitrarily, and the UI automatically scales to fit into the window. Alternatively, the window can quickly be resized to reasonable sizes by pushing *F9* (half size), *F10* (normal) and *F11* (fullscreen), respectively.

▶ Push *F9* to reduce the window size. (If nothing happens: Make sure that the window is focussed – click on its title bar.)

▶ Open a new, second terminal window, in which you run:

```
$ home2l showhouse
```

▶ Place and resize the two terminal windows and the *WallClock* window as shown in Figure 2.1.



```
    Home2L  __   ShowHouse   \ /
         ___||________     -- O --
        /   ||        \      / \
       /               \
      /_________________\
     /|  ____  ____   U |\
      | |  | |  |       |   O
      | |  | |o |       |  /I\
      | ^^^^ |  |       |  / \
--------------####--------------

(D)aylight (M)otion (L)ock (Q)uit
```

```
home2l@home2l-showcase:~$ home2l shell
home2l-shell v0.2-322-gb15d* (2018-10-06) by Gundolf Kiefer

home2l> n
server          (    127.0.0.1:4701): OK, connected (since 2018-10-06-181230)
showhouse       (    127.0.0.1:4700): OK, connected (since 2018-10-06-181230)
wallclock       (    127.0.0.1:4702): OK, connected (since 2018-10-06-181230)
home2l> l /alias/
daylight  light     lock      motion    weather
home2l> l /alias/light
/host/showhouse/building/light[bool,wr] = 0 @2018-10-06-180337.218
  ! (bool) 0     #daytime      *6   @home2l-showcase<rules:4327>/2018-10-06-180337
  ! (bool) 0     #_default     *0   @showhouse/2018-10-06-173654
  ? showhouse/UiDraw
home2l> 
```

2 Tutorial: Have a Test Drive!

2.1 Introduction

This tutorial will guide you through installing a fresh Debian Linux system in a virtual machine (which just takes very few minutes) and after that, through the installation procedure of *Home2L* in that image. As such, the tutorial can also be used as an instruction to install the *Home2L* suite on any arbitrary computer.

If you have a physical machine running Debian Linux or some derivative (other distributions may work as well, but this is untested), you can skip the installation and use your existing machine – at your choice.

i There is no pre-configured boot image available at the project page. It is not necessary. This way, the tutorial also serves as an instruction to set up an arbitrary system, which is not so difficult anyway.

2.2 Setting Up a Virtual Machine With Debian Linux

2.2.1 Prerequisites

These instructions were tested with *VirtualBox 5.2.10* and *Debian 9.5 (Stretch)*.

Up to 16 GB of disk space and 512 MB RAM will be needed for the virtual machine.

2.2.2 Creating the VM and Preparing the Installation Medium

The following steps mention Linux/Unix shell commands for illustration purposes only. Of course any OS capable of running *VirtualBox* can be used. In this case, please use the respective (graphical) tools of your OS to download, unpack and run the virtual machine.

▶ Create an empty directory 'home2l-tutorial' and change into it:

```
mkdir home2l-tutorial
cd home2l-tutorial
```

▶ Unpack the virtual machine:

Figure 2.1: Recommended Screen Layout for the Tutorial

From now on, the terminal window on the upper right is referred to as the *ShowHouse window*. The lower terminal is referred to as the *Shell window*.

i The font sizes in the terminal windows can be adapted by pushing *Ctrl–* or *Ctrl+*.

The *Home2L* suite follows a highly distributed design paradigm. At this point, there are 4 tools (called *Home2L instances*) running in your VM – 2 visible, 2 running in the background. Altogether, they simulate a typical, simple smart home setup.

The following subsections explain the running instances and their functionalities.

For experienced users:

To keep this tutorial simple, all 4 instances are running on the same (virtual) machine. Of course, they can arbitrarily be installed on multiple machines. To achieve this, the following configuration settings have be adapted:

1. In file `/opt/home2l/etc/home2l.conf`, section "Server(s)": Define the local network (`rc.network`) and allow servers to communicate over physical interfaces (`rc.serveInterface`).
2. In file `/opt/home2l/etc/resources.conf`: Edit the section "Hosts" accordingly.
3. In file `/opt/home2l/etc/home2l.conf`, section "Daemon": Setup daemon(s) to run the appropriate background processes on the respective machine(s).

2.5.1 The *ShowHouse* (`home2l-showhouse`)

The *ShowHouse* is a simulator for everything which for physical reasons cannot be provided for download: Your house and the sun. The virtual house as the following gadgets:

1. A motion sensor excited if somebody approaches the house.
2. A lockable main door with a sensor indicating whether the door is locked or not.
3. A door light, which can be switched on or off.

In reality, these would be physical gadgets accessed by respective *Home2L* hardware drivers – for example, the GPIO driver (`home2l-drv-gpio`). The *ShowHouse* is a Python script containing a *Resource* driver providing exactly the same interface internally, but lets the user simulate physical actions by key presses and visualizes the gadgets with a little ASCII art picture.

In addition to the physical gadgets, *ShowHouse* provides a resource to indicate day/night time. In a real world, this information can be obtained from the timer driver (see Section 5.9.2) based on the real time.

▶ Press the keys listed in the *ShowHouse* window and see how the various gadgets (daylight, door lock status, motion) are visualized in the ASCII art.

The outdoor light is presently controlled by a background rules script (see Section 2.5.4). It is automatically turned on for some time if motion is detected, but only at night time.

▶ Switch between daylight and night time (press 'D') and simulate motion (press 'M') to verify this behavior.

Feel free to inspect the source code of the *ShowHouse* (`/opt/home2l/bin/home2l-showhouse`). It is a Python script and may contain a number of useful code examples for writing own rules, drivers or applications.

2.5.2 The *WallClock* (`home2l-wallclock`)

`home2l-wallclock` is the main end user interface – a universal information display to be mounted on the wall of rooms or installed on mobile devices.

The *WallClock* shows the current time and date in its upper area. The bottom row contains launcher buttons to access the integrated calendar and music applets (to be explored later).

Most relevant for this part of the tutorial is the lower part of the display between the date and the launcher buttons, in which the values of several sensors – *Home2L resources* – are visualized.

These are – from left to right (referring to Figure 2.1):

1. Status of the door lock.
2. A motion indicator.
3. The *radar eye* – a visualization of weather radar around the building (the location is pre-configured for Augsburg, Germany).
4. Air pressure and humidity.
5. Local outside temperature.

Each of these fields visualizes some *Home2L resource* as selected in the "WallClock" section of `/opt/home2l/etc/home2l.conf`. The resources are provided from different sources.

The *Resources* library can transparently deal with the absence of resource data. For example, if the weather driver is absent or cannot establish an internet connection to the weather data server, the weather-related displays will simply disappear.

i

Troubleshooting

The weather driver depends on an available internet connection and on the *OpenData* server of the German Weather Service (DWD). If the server is not reachable at the time you run the tutorial, the weather-related displays mentioned above will be missing. If so: Do not worry, the other parts of the tutorial are not affected.

You can diagnose the availabilty of weather data by running the *Weather* driver directly in a terminal:

```
$ .  /opt/env.sh
$ /opt/home2l/lib/home2l-drv-weather weather.debug=1
```

▶ In the *ShowHouse* window, push 'M' and observe the motion display in the *WallClock* window.

▶ In the *ShowHouse* window, push 'L' multiple times and see that the door lock display in *WallClock* window instantaneously indicates the current lock status.

▶ In the *ShowHouse* window, unlock the door (push 'L' as necessary). Then toggle the daylight status (push 'D'). In the *WallClock* window, the lock icon is highlighted if the door is unlocked at night time to remind the user to keep it locked over night.

▶ Quit the *ShowHouse* (push 'Q') and see the lock icon disappearing in the *WallClock* display. Start the *ShowHouse* again (push cursor up and return) and see the lock icon re-appearing.

Repeat this to see that the *WallClock* recognizes the presence/absence of resources and servers instantaneously!

2.5.3 A Server (`home2l-server`)

This is an instance of the background server (`home2l-server`), which in this tutorial hosts the weather driver (`home2l-drv-weather`).

i

In general, *Home2L* does not necessarily need a central server. In this example, the weather driver may as well be loaded and executed by *any* of the other three *Home2L* instances.

In practice, however, it may be helpful to distribute the drivers over multiple *Home2L* server instances to improve failure resilience, to connect sensors/actors to different machines, or for maintenance reasons.

The assignment of the *weather* driver to the *server* instance in configured in the `home2l.conf` configuration file and may be changed there.

2.5.4 A Rules Script (`rules-showhouse`)

There is presently one automation script active with rules to

a) automate the outdoor light,

b) keep the *WallClock* on, preventing it from auto-switching off.

This script will be explored later in Section 2.9.

▶ Shut down the *Home2L* background services:

```
$ sudo systemctl stop home2l
```

and see that

a) the weather information disappears from the *WallClock* display,

b) the outdoor light no longer switches on automatically on motion at night.

▶ Restart the background services

```
$ sudo systemctl start home2l
```

and verify that the weather display and the outdoor light are working properly again.

Congratuilations! You have now moved in succesfully into your new house and tested its basic functionality. Next, we will take a look behind the scenes.

2.6 Using the *Home2L Shell*

This section gives an introduction to the *Home2L Shell* (home2l-shell), the main command line interface, which we will now use to take a look behind the scenes. Im particular, we will use the shell to check the status of all *Home2L* hosts, explore the directory tree, manipulate and monitor resources, and use the shell for logging purposes.

▶ Go to the *Shell window* and start the *Home2L Shell*:

```
$ home2l shell
```

The shell has an integrated help functionality. Type

```
home2l> h
```

to get a list of available commands or

```
home2l> h <command>
```

to get more detailed information about a certain command.

▶ Check the status of all servers in the *Home2L* network:

```
home2l> n
server          (    127.0.0.1:4701): OK, connected (since 2018-10-28-194909)
showhouse       (    127.0.0.1:4700): OK, connected (since 2018-10-28-194909)
wallclock       (    127.0.0.1:4702): OK, connected (since 2018-10-28-194909)
```

The list indicates that all servers are OK and reachable from the *Home2L Shell*.

2.6.1 Navigating and Inspecting Resources

▶ Use the commands 'c' and 'l' to explore the available resources and the namespace. List the root directory of the namespace:

```
home2l> l /
alias
env
host
local
```

The directory /host contains all hosts:

```
home2l> l /host/
home2l-showcase<shell:2254>
server
showhouse
wallclock
```

The directory /alias contains all defined alias names, which are symbolic links to host resources or directories:

```
home2l> l /alias/
daylight -> /host/showhouse/building/simDaylight
light -> /host/showhouse/building/light
lock -> /host/showhouse/building/lock
motion -> /host/showhouse/building/motion
weather -> /host/server/weather
```

The top-level directory /local is a hard-wired alias to /host/<self>, where <self> is the local *Home2L* host ID.

▶ With the 'c' command, you can navigate in the tree. Of course, tab-completion is available, too.

```
home2l> c /alias/light
/alias/light
```

i If you are familiar with navigating and exploring files using 'cd', 'ls' and tab-completion in a Unix shell: Navigating with the *Home2L Shell* is very similar, just easier!

a) The respective commands have one character instead of two ('c'/'l' instead of 'cd'/'ls').

b) Tab-completion may auto-complete over multiple directory levels.

c) The 'l' command does not only list directories, but displays anything (particularly resources, as the next example shows). Simililarly, with 'c' you can not only navigate to a directory, but also to a resource itself.

▶ The 'l' command without arguments lists the current directory or object. Running it now (current location is /alias/light) displays the outdoor light resource.

```
home2l> l
/host/showhouse/building/light[bool,wr] = 0 @2018-10-28-193047.599
  ! (bool) 1     #motion      *5 -2018-10-28-195452.133   @home2l-showcase<rules↩
   ↪:498>/2018-10-28-195447
  ! (bool) 0     #daytime     *6   @home2l-showcase<rules:498>/2018-10-28-193047
  ! (bool) 0     #_default    *0   @showhouse/2018-10-28-193045
  ? showhouse/UiDraw
```

The first line of the output shows the URI (unified resource indicator) of the resource, its type, writability, current value and the time of the last change. The next lines list the active requests and subscribers.

Lines starting with an "!" show an active request. In the example above, there is a request with the ID '#_default' for a value of 0 with a low priority ('*0') and another one '#motion' for a value of 1 with a higher priority, which superseeds the default request in this situation. The '#motion' request has a time-out attribute set causing the request to be auto-removed at the given time. The '@' tags at the end of the lines identify the origin (host/time) for each request.

> **i** This example reflects the situation shortly after pushing the "M" button in the *ShowHouse* window at (simulated) daytime. The '#motion' request tries to switch on the light for 5 seconds (until 19:54:52). However, a '#daylight' request at a higher priority is active as well, which ties the value to 0, so that the light effectively remains off.

Lines starting with a "?" show a subscriber. In this example, there is one subscriber, namely the `'UiDraw()'` function of the *ShowHouse* script, which is responsible for redrawing the ASCII art on value changes.

- Push "D" and "M" in the *ShowHouse* to change the daylight status and simulate motion (or not) and repeat the above command in various situations to see how the respective requests change.
- Explore the directory tree of the installation! Find out, which resources available from which of the three *Home2L* server instances!

2.6.2 Manipulating Resources Manually

We now manipulate the light by placing requests manually.

- Enter (*Note: the second command is the digit "one", not a lower-case"L"*)

```
home2l> c /alias/light
home2l> 1
/host/showhouse/building/light[bool,wr] = 1 @2018-10-28-201230.940
  ! (bool) 1     #shell        *8   @home2l-showcase<shell:2254>/2018-10-28-201230
  ! (bool) 0     #daytime      *6   @home2l-showcase<rules:498>/2018-10-28-193047
  ! (bool) 0     #_default     *0   @showhouse/2018-10-28-193045
  ? showhouse/UiDraw
```

  and see that the light is turned on. Look at the output: Can you identify the request you created with the "1" command?

- Make sure that night mode is set in the *ShowHouse* window (push "D" as necessary). The light will remain on. Then enter

```
home2l> 0
/host/showhouse/building/light[bool,wr] = 0 @2018-10-28-202750.949
  ! (bool) 0     #shell        *8   @home2l-showcase<shell:2254>/2018-10-28-202750
  ! (bool) 0     #_default     *0   @showhouse/2018-10-28-193045
  ? showhouse/UiDraw
```

  This forces the light to stay off. Motion events have no effect on the light even at night, since their requests have a lower priority ("*6") than the default priority of shell requests ("*8").

- Finally, remove the manual shell request by typing

```
home2l> -
/host/showhouse/building/light[bool,wr] = 0 @2018-10-28-202906.524
  ! (bool) 0     #_default     *0   @showhouse/2018-10-28-193045
  ? showhouse/UiDraw
```

i The commands "0", "1", and "-" are abbreviations for variants of the "r+" and "r-" commands to place and remove requests. More information on this is given in the online help.

▶ It is possible to pass arbitrary request attributes (see Section 5.6) as command line arguments. To turn on the light in 2 seconds and off again 3 seconds later enter:

```
home2l> 1 +2000 -5000
/host/showhouse/building/light[bool,wr] = 0 @2018-10-28-202906.524
  ! (bool) 1     #shell        *8 +2018-10-28-203449.268 -2018-10-28-203452.268   ↩
    ↪@home2l-showcase<shell:2254>/2018-10-28-203447
  ! (bool) 0     #_default     *0   @showhouse/2018-10-28-193045
  ? showhouse/UiDraw
```

The arguments '+2000' and '-5000' add start and stop time attributes to the request so that it becomes effective in 2000ms from now until 5000ms from now.

▶ (Optional) Explore the directory tree and watch out for writable resources!

Can you find out how to dim or switch off the *WallClock* display?
(Hints on this are given below in Section 2.9.)

Try manipulating the resources, but afterwards, remove all requests again you have set in this exercise ("r-" command).

2.6.3 Monitoring Resources

The shell allows to subscribe to any resources using the "s+" and "s-" commands, respectively.

▶ Subscribe to all resources provided by the *ShowHouse* building:

```
home2l> s+ /host/showhouse/building/*
Subscriber 'home2l-showcase<shell:7336>/shell'
  /host/showhouse/building/*?
  /host/showhouse/building/light
  /host/showhouse/building/lock
  /host/showhouse/building/motion
  /host/showhouse/building/simDaylight
: /host/showhouse/building/light = ?
: /host/showhouse/building/lock = ?
: /host/showhouse/building/motion = ?
: /host/showhouse/building/simDaylight = ?
```

The first part of the output (without the lines starting with ":") shows the ID of the subscriber of the shell and lists the resources it currently subscribes to. Lines ending with "?" are *watch set* entries and are currently not associated with an existing resource.

> *Watch set* entries may either be named resources or – as in this case – wildcard patterns. They allow to start and stop the subscribing and serving hosts independently from each other. You can verify this by closing both the *ShowHouse* and the *Home2L Shell*, and then repeating the above command *before* starting the *ShowHouse*.
>
> The command "s" lists the status of the shell's subscriber. Run it before and after starting the *ShowHouse*.

Lines starting with a colon (":") are events reported by the subscriber. The *Home2L* suite follows a very precise event model. Immediately after the subscription, the locally known values are reported ("unknown" in this case). However, in the background, the server(s) contacted. By the time you have read this text, the actual values have been delivered from the server(s) to the shell.

▶ Press return:

```
home2l>
: /host/showhouse/building/light = 0 @2018-11-01-184558.651
: /host/showhouse/building/light connected
: /host/showhouse/building/lock = 0 @2018-11-01-184558.651
: /host/showhouse/building/lock connected
: /host/showhouse/building/motion = 0 @2018-11-01-184558.651
: /host/showhouse/building/motion connected
: /host/showhouse/building/simDaylight = 0 @2018-11-01-184558.651
: /host/showhouse/building/simDaylight connected
```

Incoming events are collected in the background and displayed after each shell command.

▶ To follow and display events immediately when received, run the "follow" command:

```
home2l> f
```

▶ Now make some interactions in the *ShowHouse* window (e.g. lock/unlock the door, switch to daytime and move the person). The output will be similar to this:

```
: /host/showhouse/building/simDaylight = 1 @2018-11-01-184833.883
: /host/showhouse/building/lock = 1 @2018-11-01-184841.820
: /host/showhouse/building/lock = 0 @2018-11-01-184846.884
: /host/showhouse/building/simDaylight = 0 @2018-11-01-184848.494
: /host/showhouse/building/motion = 1 @2018-11-01-184849.757
: /host/showhouse/building/light = 1 @2018-11-01-184849.759
: /host/showhouse/building/motion = 0 @2018-11-01-184850.257
: /host/showhouse/building/light = 0 @2018-11-01-184854.761
```

▶ Press *Ctrl-C* to stop "follow" mode. Events for subscribed resources are still reported with their correct time stamps, but only after commands have been entered.

▶ Quit the shell by pressing *Ctrl-D*.

2.6.4 Non-Interactive Use and Resource Event Logging

The *Home2L Shell* can also be run in a non-interactive way, so that certain actions can be performed programmatically from shell scripts.

- For example, this command turns on the *ShowHouse* light for 2 seconds:

  ```
  $ home2l shell -e "c /alias/light; 1 -2000"
  ```

- To log all motion events, you may use a command like:

  ```
  $ home2l shell -e "s+ /alias/motion; f" > motion.log & PID=$!
  ```

  To stop logging, enter:

  ```
  $ kill $PID
  ```

- To log the outside temparature over time, run:

  ```
  $ home2l shell -e "s+ /alias/weather/temp; f"
  ```

  (Again, this job may optionally be backgrounded and its output redirected as in the previous example.)

2.7 *WallClock* Gadgets: The Music Player

The *WallClock*'s music player is a Music Player Daemon (MPD) client, optimized for a home setup with multiple *WallClocks* in different rooms and multiple music machines. For this tutorial, the server has been already installed and started during the previous steps.

- Copy some of favorite *mp3* files into

  ```
  /var/opt/home2l/mpd/music
  ```

  Alternatively, you can download some free music from `https://freemusicarchive.org`, for example.

- Update the *MPD* database

  ```
  $ mpc -h localhost update
  ```

- Put the *WallClock* window into fullscreen mode (push *F10* or *F11*) and push the *"Music"* button.
- With the player selection button on the bottom, select *"ShowStage"*.
- In the navigation pane (right half of the screen), navigate to your favorite song. Pushing the title bar of the navigation pane will navigate up or switch between the local database and playlists.
- Select your favorite song, push the play button and enjoy!

Troubleshooting

The integrated music player relies on a working MPD (should have been installed so far) and a working audio output of your virtual machine (VM).

The MPD configuration file is located in `/opt/home2l/etc/mpd-showstage.conf`. It should work out-of-the-box, but may (have to) be further adapted to your preferences. By default, the audio output is directed to the default ALSA playback channel.

To test if audio playback is working inside your VM, run:

```
$ speaker-test -t sine
```

To adjust the volume, run:

```
$ alsamixer
```

To update the music database after you have copied files into the music directory:

```
$ mpc -h localhost update
```

2.8 *WallClock* Gadgets: The Family Calendar

The calender applet uses `remind(1)` as a backend and thus supports its syntax for specifying events.

- Put the *WallClock* window into fullscreen mode (*F11* or *F10*) and start the applet by pushing "Calendar" on the main screen.
- Explore the calendar UI by navigating to different dates (e.g. your wife's next birthday or your next anniversary).
- Select an event in the right pane and modify it (e.g. change its time or text).
- Select a day in the left pane and add a new one-time appointment for Julian, for example (enter/keep your own date at the beginning):

  ```
  2018-12-23 at 19:00 dur 2:00 REM Meet Henry; Murphy's Pub
  ```

2.9 Writing Automation Rules

This section introduces the development of automation rules. *Home2L rules* are normal Python scripts. They can be tested separately from already existing rules files and later be merged into them.

- Make sure that the *WallClock* window is small (push *F9*) and the screen layout is as sketched in Figure 2.1. The *ShowHouse* must be running, and in the *Shell window*, a normal command prompt must be active (quit the *Home2L Shell*, if it is still running).
- Inspect and read the supplied rules file:

```
$ sudo nano /opt/home2l/etc/rules-showhouse
```

Can you identify the rules for controlling the light?

What would you need to change to increase the light-on duration after a motion event? How is the light kept off at day time?

▶ Identify the rule for keeping the *WallClock* permanently active

```
@daily("wallclock")
def PermanentRules (host):
  ...
```

and deactivated it by commenting out the first line with the `@daily` decorator:

```
# \@daily("wallclock")
def PermanentRules (host):
  ...
```

Finally, exit `nano` and save the file (*Ctrl-X*).

▶ To make this change effective, restart the modified rules script `rules-showhouse` and remove its permanent request:

```
$ pkill rules-showhouse
$ home2l shell -e "c /host/wallclock/ui; r- active python; r- standby rules"
```

After some time (at most 10 seconds, see the `ui.standbyDelay` setting in `home2l.conf`), the *WallClock* UI should dim or turn off completely. Clicking on it re-activates it for some time. You may test this now, but should then wait again until the screen dims. After that, you should not click into the *WallClock* window anymore.

i Like potentially any *Home2L* application, the *WallClock* exports a number of resources. For example, on *Android* it can export the device's brightness sensor value or Bluetooth status. In this tutorial, we will use `ui/active`, `ui/standby` for controlling the UI state and optionally `ui/mute` to mute the audio player.

▶ Setup all necessary environment variables and start an interactive Python session:

```
$ . /opt/home2l/env.sh
$ python3
```

Inside this session, we will now develop a new rule to control the active/standby state the *WallClock* application.

i Instead of working in the interactive shell, you can, of course, enter the Python commands of the following instructions into the `rules-showhouse` file or into a new file. A new file can be generated as follows:

```
$ echo '#!/usr/bin/python3' > myrules $ chmod a+x myrules
```

This is particularly useful if you intend to experiment with the rules and change them.

Enter all following instructions at the Python command prompt (`>>>`). (The prompt is not always printed in the following instructions to facilitate copy-and-paste.)

▶ Import the *Home2L* package and initialize it:

```
from home2l import *
Home2lInit ("myrules")
```

▶ Get references to all resources used later:

```
rcDaylight = RcGet ("/alias/daylight")
rcLock = RcGet ("/alias/lock")
rcUiActive = RcGet ("/host/wallclock/ui/active")
```

▶ (Optional) In an interactive Python shell, you can now inspect the network environment and resources:

```
>>> Home2lStart ()    # Start the Home2L background tasks
>>> RcHosts ()
['home2l-showcase<python:6862>', 'server', 'showhouse', 'wallclock']
>>> RcHostResources ("wallclock")
['/host/wallclock/timer/daily', '/host/wallclock/timer/hourly', '/host/wallclock/↩
    ↪timer/minutely', '/host/wallclock/timer/now', '/host/wallclock/timer/twilight/↩
    ↪dawn06', '/host/wallclock/timer/twilight/dawn12', '/host/wallclock/timer/↩
    ↪twilight/dawn18', '/host/wallclock/timer/twilight/day', '/host/wallclock/timer/↩
    ↪twilight/day06', '/host/wallclock/timer/twilight/day12', '/host/wallclock/timer↩
    ↪/twilight/day18', '/host/wallclock/timer/twilight/dusk06', '/host/wallclock/↩
    ↪timer/twilight/dusk12', '/host/wallclock/timer/twilight/dusk18', '/host/↩
    ↪wallclock/timer/twilight/sunrise', '/host/wallclock/timer/twilight/sunset', '/↩
    ↪host/wallclock/ui/active', '/host/wallclock/ui/bluetooth', '/host/wallclock/ui/↩
    ↪bluetoothAudio', '/host/wallclock/ui/dispLight', '/host/wallclock/ui/luxSensor'↩
    ↪, '/host/wallclock/ui/mute', '/host/wallclock/ui/standby']
>>> rcDaylight
(CResource) /host/showhouse/building/simDaylight bool ro
```

Of course, online help and tab-completion is available for all commands. For example, try:

```
>>> help (RcGet)
Help on function RcGet in module home2l:

RcGet(uri, allowWait=False)
    RcGet(char const * uri, bool allowWait=False) -> CResource
    RcGet(char const * uri) -> CResource

    Lookup a resource by its URI and return a reference to it.
```

▶ Define a rule which switches on the display whenever the door is unlocked at night (mind the indentation!):

```
@onUpdate(rcDaylight, rcLock)
def UpdateUiState ():
  print ("\#\#\# daylight = " + str (rcDaylight.ValueState ()) + ", lock = " + str (↩
    ↪rcLock.ValueState ()))
  daylight = rcDaylight.ValidValue (False)
  doorLocked = rcLock.ValidValue (False)
  if not daylight and not doorLocked:
    rcUiActive.SetRequest (value = True)
  else:
    rcUiActive.DelRequest ()
```

The first line ("`@update()`"), a Python *decorator*, makes the function `UpdateUiActive()` to be called whenever the value of any of the supplied resources changes. Inside the function body, the values of `rcDaylight` and `rcLock` are read using the `ValidValue()` method, which takes a default value as an argument and always returns a valid value, even if the actual resource value is currently unknown. Depending on these values, a request is set to switch the display active or not.

- (Optional) Define another rule to mute the music player for 5 seconds if motion is detected in front of the house:

```
@onEvent ("/alias/motion")    # TBD: Test
def MuteOnMotion (ev, rc, vs, data):
  if ev == rceValueStateChanged and vs.ValidValue (False) == True:
    RcSetRequestFromStr ("/host/wallclock/ui/mute", "1 -5000")
```

For demonstration purposes, this example uses some other API functions than the previous ones:

  – Instead of `@onUpdate()`, the `@onEvent()` decorator is used, which allows to precisely track all events.

  – The resources are directly identified by their URI without first getting an object reference by `RcGet()`.

  – The request is set by `RcSetRequestFromStr ()`, which allows to describe the request by a string with the same syntax as already known from the *Home2L Shell*.

Details can be found in the Python API and in Section 5.7.

- Run the *Home2L* main loop:

```
Home2lRun()
```

The new rule(s) are now active.

- Test the new rule(s) by interacting with the *ShowHouse*!

2.10 Going Further

To learn more about the capabilities of the *Home2L* suite, we suggest to look into the configuration files and source code of the tutorial.

- The supplied rules file has already been inspected in Section 2.9:

```
$ nano /opt/home2l/etc/rules-showhouse
```

- Inspect and read the *ShowHouse* source file:

  ```
  $ nano /opt/home2l/bin/home2l-showhouse
  ```

  It contains an example for a resource driver implemented in Python, namely for the *ShowHouse* gadgets and the keyboard input. The latter is a bit more sophisticated and involves a background thread.

  The ASCII art visualization is refreshed whenever necessary, but not unnecessarily often. This is achieved by implementing the drawing function as a rule.

- Inspect or read the main configuration file

  ```
  $ nano /opt/home2l/etc/home2l.conf
  ```

Feel free to modify any of these or other files to learn more about their options. If you change a configuration file, it may be necessary to restart the *Home2L* background services for the changes to take effect.

If you are using the *systemd* init system (as in the *Home2L ShowCase* VM used here), the *Home2L* Daemon with all its services is restarted by the following command:

```
$ systemctl restart home2l
```

If you are using the *System-V* init system, the *Home2L Daemon* is restarted by:

```
$ service home2l restart
```

A shortcut to restart just a single instance – for example, the *home2l-rules* script – is to simply kill that instance. It will be restarted automatically by the *Home2L Daemon*.

```
$ pkill home2l-rules
```

3 Compiling and Installing

3.1 Overview

Home2L comes with its own build system. This is accounts for the fact that *Home2L* is not a piece of software to be installed on a single computer, but instead, the *Home2Ls* are installed in a "home", which is a heterpgeneous *cluster* of machines with different hardware (e.g. x86, ARM) and operating software environments.

The build system is therefore capable for cross-compilation. Presently, it assumes a Debian-based environment and is tested under Debian 9.x (Stretch).

3.2 Prerequisites

The requirements for building the core part of the *Home2Ls* are quite small:

- A C/C++ compiler (compliant with the *C99* and *C++11* standards) with basic libraries (`libc`, `libstdc++`).
- *Python 3* with development packages and *SWIG* (>= 3) for the *Python API*.
- `libreadline` (optional, for the `home2l-shell`).

Section 2.3 provides an exact list of packages to install on Debian or Debian-based systems.

Building the documentation (module '`doc`') requires a couple of additional packages (`doxygen, texlive, graphviz, ...`). Most users do not have to build the documentation, since readable versions are available on the project page.

3.2.1 Cross-Compilation

Cross-compilation is supported by the *Home2L* build system based on the Debian cross-building capabilities for the architectures `i386`, `amd64`, and `armhf`. The development machine must have `i386` or `amd64` as its primary architecture. The other architectures must be entered as additional architectures to the `dpkg(1)` package manager

To cross-build `armhf` binaries on an `i386` or `amd64` machine, as of Debian 9 ("Stretch"), the following packages must be installed:

```
crossbuild-essential-armhf g++-6-multilib
```

For all desired target architectures, the respective development packages mentioned above must be installed. .

3.2.2 Optional External Libraries

Some applications require additional external libraries, sometimes depending on the options they are compiled with, as indicated in Figure 1.1. The folder `externals/` in the source tree contains hints on how to obtain, build and setup the respective libraries for different platforms. Please note, that this folder is distributed "as is" without any warranty and may potentially be incomplete. Files to watch for are:

`prebuild.sh`: A build script with comments on how to obtain the sources and hints for building.

`Debian.mk`: A Debian/Linux makefile fragment.

`Android.mk`: An Android NDK makefile fragment.

3.3 Compiling

To build and install the suite, do the following:

1. View the build options by running the following command in the main source directory to view the build options:

   ```
   $ make help
   ```

2. Check and, if necessary, adapt the compiler and build settings in file `Setup.mk`.
3. Build:

   ```
   $ make <options>
   ```

4. Install the *blob* to `$HOME2L_ROOT` (default: `/opt/home2l`):

   ```
   $ make <options> install
   ```

This will install the so-called *Home2L blob* on your computer. It contains all files for all architectures together with a sample configuration (in `$HOME2L_ROOT/etc/`). This *blob* can now be simply copied to all machines of your cluster. The `home2l-rollout` tool can automate that for software or configuration updates. To use the *Home2Ls*, some additional things have to be set up. This is explained in the following sections.

3.4 Setting Up Users and Permissions

3.4.1 Users and Groups

The following users are involved and must exist on each machine:

- User 'home2l' with primary group 'home2l': Under this UID, all *Home2L* background processes are executed. This user should get all permission required for its background or end-user tasks, but no more than that. In particular, 'home2l' should not be allowed to modify configuration files.
- User 'root': The super user.
- The user account of the administrating user - we call her 'myadmin' here.

The user 'home2l' must have 'bash' as its login shell and the following line in its .bashrc file to have all 'HOME2L_*' environment variables set when required:

```
source $(home2l -e)
```

3.4.2 Automatic ssh Logins sudo Rules for Cluster Administration

For central cluster adminstration using home2l-rollout, the following *sudo* rules are required on each machine of the cluster:

- for 'myadmin': run home2l-install as 'root'
- for 'myadmin': run adb(1) as 'home2l' (only on machines hosting Android devices)
- for 'home2l': run home2l-sudo as 'root' (optional)

The following *ssh* logins must be possible without a password in the cluster ("master" is the master machine):

- from 'myadmin@master' to any other machine as user 'myadmin',
- from 'root' at any non-master host to 'home2l@master'.

3.4.3 File Permissions

The file permissions in the installation directory (including etc/) are maintained by the tools home2l-install and home2l-rollout as follows:

- home2l-install sets the ownership to 'root:home2l'. Permissions are preserved from the master, 'make install' sets them to 644 for files and 755 for directories.

- The folder etc/secrets is meant for storing sensitive data only readable by members of the group 'home2l'. The permissions are set to 640 for files and 750 for directories. Hence, only 'root' or users of group 'home2l' can read them, and only 'root' can modify them. Others have no access.

3.5 Adding a New Machine to the Cluster

To install the *Home2L* suite on the first computer, follow the building and installation steps described Section 3.3 to install a *Home2L blob* on your *master computer*. To complete the installation, run (assuming that the blob is installed in /opt/home2l):

```
$ sudo /opt/home2l/bin/home2l-install -i
```

The tool explains itself what it is doing.

To prepare a new additional (Linux) machine and add it to the cluster, the installation blob can be cloned from an existing (typically the *master*) machine.

1. On the *master*: Edit the configuration files rollout.conf, install.conf and init.conf to reflect the new setup with the new machine.
2. On the new machine: Setup users and their rights as described in Section 3.4.
3. On the new machine: Replicate the blob from the master by running a command like
   ```
   $ sudo rsync -va --perms --chown=root:home2l home2l@master:/opt/home2l/ /opt/↩
       ↪home2l
   ```
4. On the new machine, run:
   ```
   $ sudo /opt/home2l/bin/home2l-install -i
   ```
5. On the master, run
   ```
   $ home2l-rollout
   ```
   and check, if the new machine is listed. Press 'y' ("yes") to run the rollout procedure and see if updating the new machine works without errors.

From now on, software and configuration updates can be rolled out by the tool home2l-rollout from the master.

3.6 Installing the *WallClock* on Android

This section is under construction.

Compiling and installing the *Android* app presently requires good knowledge in programming and *Android* debugging. The following text is only a brief sketch on what to do (see Section 1.3).

To set up an *Android* hosts, for example a *WallClock* tablet, the following steps have to be done:

1. *Not strictly necessary, but recommended:* Install a Linux distribution environment on the Android device (e.g. using *Lil'Debi* or *Linux Deploy*). Then install *Home2L* into this environment as described in Section 3.5. Install the blob to `/data/home2l` (not `/opt/home2l`). This allows to use the same blob from the app and the Linux distribution environment.

   This environment allows you to:

   - use the *WallClock* calendar applet,
   - see the system info on the *WallClock* system info screen,
   - use `home2l-rollout` to maintain this machine,
   - use the *Home2L* command line tools in this environment.

2. *If you do not have a Linux distribution environment (previos step):* Copy the *Home2L blob* to `/data/home2l` (e.g. using `adb(1)`).

3. Install the *WallClock* Android app.

4. For some of the functionality mentioned in step 1, the Android app needs to run commands in the Linux distribution environment. This is done by SSH logins from the Android environment into the Linux distribution environment. To set this up:

   a) Generate an SSH identity without a password and store it as `'etc/secrets/ssh/<hostname>[.pub]'`.

   b) In the Linux distribution environment, install an SSH daemon and allow logins without password from `'localhost'` as user `'home2l'`.

   c) Test the connection from an Android shell:

   ```
   $ adb shell
   $ su   # to make etc/secrets/ssh/* accessible
   # ssh -i /data/home2l/etc/secrets/ssh/mymachine  -o ↩
       ↪NoHostAuthenticationForLocalhost=yes home2l@localhost
       # Replace 'mymachine' with the hostname of your Android machine.
       # A shell in the Linux distribution environment should open.
   mymachine: $ $HOME2L_ROOT/bin/h2l-sysinfo.sh
       # This should print a list of processes and further system information.
   ```

   Similar to what is mentioned in Section 3.4, `home2l-install` maintains the file permissions in `etc/secrets` in a special way, with the following additions on Android:

   - The ownership is set to the UID of the *Home2L WallClock* Android app.

- The folder 'etc/secrets/ssh' and its contents are set group-inaccessible, so that nobody but the *WallClock* app owner (and root) can read the SSH IDs.

If the *WallClock* app fails to start due to some configuration problems, detailed information for diagnosis can be found in the Android log system. This can be viewed using adb(1):

```
$ adb logcat -v time home2l:D SDL:V *:E *:W *:I
```

4 Managing a *Home2L* Installation

A typical *Home2L* installation is distributed over multiple computers (*machines*), on each of which one ore multiple *Home2l instances* may be running. There is no central server, all *Home2L instances* are equal in rank.

4.1 Terminology

In this document, the following terminology is used:

A *machine* is a physical computer.

A *Home2L instance* is a running program (process) using the *Home2L* library, such as `home2l-wallclock`, `home2l-shell`, or a rules script. Each instance is identified by its ***instance name*** (or *instance ID*), which is typically the name of its executable (without a leading "home2l-"), but may be set to a different value by the respective application (see `sys.instanceName`). The instance name must be unique on a machine.

A *Home2L cluster* (or *Home2L network*) is a set of machines running *Home2L* instances that interact with each other and make up the *Home2L* installation.

The *master* machine is the computer from which a cluster is managed, typically the desktop PC of the adminstrator. From this machine, software and configuration updates are rolled out.

A *Home2L server* is an instance exporting *Home2L* resources (see Chapter 5).

A *Home2L client* is an instance accessing resources (i.e. any process incorporating the *Home2L Resources* library).

A *Home2L host* a *Home2L* server or client, in other words: an instance communicating in a *Home2L cluster*. It is identified by the ***host ID*** as declared in `resources.conf`.

i

The term *'host'* is frequently used as a synonym for a computer or machine. However, its meaning may as well differ slightly. For example,

a) a *host name* may refer to an IP address in a network, and as such identify an interface of a computer, not the computer itself, which may have multiple interfaces.

b) the term *host* may refer to the operating system running on physical hardware as opposed to the guest operating sytem running in a virtual machine on the same hardware.

In the *Home2L* context, a *host* actually refers to a *Home2L* instance running on a machine. Often, there is only one instance running per machine, so that its host ID is equivalent to the machine name (aka "hostname"). However, there may as well be multiple servers with different *host IDs*.

To avoid ambiguities, this book avoids to use the term *host* for anything other than a *Home2L host*, even if it is common to call a machine "host", speaking of a "hostname" as the name identifying a machine etc. .

4.2 Common Maintenance Tools

The common tools for maintaining *Home2L* installations are:

- **`home2l-shell`**: The command line interface and "swiss army knife" to access and inspect the *Home2L* resources. Details can be found in Section 5.8.
- **`home2l-rollout`**: Tool to distribute configuration changes or software updates from the master to the other machines.
- **`home2l-install`**: Internal tool for performing various installation tasks on the local machine. This tool is usually not called manually, but indirectly by `home2l-rollout`.
- **`home2l-sudo`**: (optional) Container to allow the `home2l` user to perform a limited set tasks with `root` privileges (e.g. to restart certain system services if they fail to ensure long-term availability). The use of this tool is optional, to use it, the file `/etc/sudoers` has to be set up such that `home2l` can run it as `root`. The allowed activities are encoded in the tool itself. Please note, that the tool is presently implemented as a shell script and therefore prone to (yet unknown) potential security holes.
- **`home2l-adb`**: (optional) Wrapper for the *Android Debug Bridge (ADB)*, allows to maintain Android machines connected by USB cable to Linux machines (requires `androidb.conf` to be set up).

Each of the tools implements a `-h` (help) option, which gives up-to-date usage information.

Tools can be invoked in one of two ways:

a) Calling the tool directly after setting the *Home2L* environment settings, for example:

```
$ source /opt/home2l/env.sh     # This line may be put into .bashrc .
$ home2l-shell
```

b) Using the general invoker without source'ing `$HOME2L_ROOT/env.sh` first, for example:

```
$ home2l shell
```

The script named `home2l` sets the environment variables and calls the tool passed as arguments. To make this work, a symbolic link to `$HOME2L_ROOT/bin/home2l` must exist somewhere in the search path (e.g. in `/usr/local/bin`)

4.3 Common Configuration Files

A complete *Home2L cluster* is configured by a single set of configuration files, which is are stored and maintained on each machine of the cluster in a synchronized way. Whenever changes are necessary, the administrator edits the configuration on the master and replicates changes to the other machines by calling `home2l-rollout`.

i This strategy of replicated configuration files is motivated by nature: Each individual cell of an animal or plant on earth contains an identical copy of the complete DNA set of the respective species. In nature, this appears to be a good strategy for quite some million years of evolution now ...

The commonly relevant configuration files are:

- **`home2l.conf`**: Main configuration file (see Section 4.4).
- **`resources.conf`**: Information for the *Resources* library (see Section 5.4).
- **`rollout.conf`**: Declaration of machines in the cluster.
- **`install.conf`**: Installation options of machines in the cluster.
- **`init.conf`**: Configuration for the *Home2L* init script and daemon.
- **`androidb.conf`**: (optional) Declaration of all Android devices, if `home2l-adb` is used to manage them from the master.

The syntax and contents of the main configuration file and the *Resources* configuration are explained in Sections 4.4 and 5.4, respectively.

For the other files, explanations can be found as comments inside the files themselves. A set of sample configuration files is automatically installed with a new installation.

4.4 Main Configuration File Syntax

4.4.1 Overview

The main configuration file is expected as `$HOME2L_ROOT/etc/home2l.conf`. An alternative file can be specified by the `HOME2L_CONFIG` environment setting or the `-c` command line option of most tools.

All *Home2Ls* and all applications linked against the *Home2L* library have access to parameters defined there via the `EnvGet()` and `EnvGet<type>()` API calls.

The syntax is based on that of INI files. The file contains a set of lines of parameter assigments in the format:

```
<key> = <value> [ ( ";" | "#" ) <comment> ]
```

The characters ';' and '#' start a comment, everything behind them is ignored. A value may optionally be quoted by single(') or double (") quotes. From unquoted values, leading and trailing whitespaces are stripped. A backslash (\) can be used to escape a single character.

Examples:

```
example.someInt = 17              # some integer value
example.string1 = Hello world!    # a string with 12 characters
example.string2 = "  space  "     # a string with 9 characters
example.string3 = ' ";" '         # a string with the comment and double-quote character
example.string4 = ' \"\' \" '     # a string with 6 characters using both types of quotes
example.bool1 = 0                 # Boolean value, equivalent to '-', 'F', 'f', ...
                                  # ... 'false', 'False', or 'FALSE'
example.bool2 = true              # Boolean value, equivalent to '1', '+', 'T', 't', ...
                                  # ... 'true', 'True', or 'TRUE'
```

Keys are case-sensitive. Additional parameter settings can be specified by the `HOME2L_EXTRA` environment variable (which precedes over settings in the configuration file) and as command line parameters for most tools (which would precede over the configuration file or `HOME2L_EXTRA` settings).

For example, the following command runs the *Home2L Server* application with additional debug output:

```
$ home2l server debug=1
```

The contents of the main configuration file can be split over multiple files. The following assignemt has a special meaning and acts as an *include* directive. Multiple and nested includes are possible.

```
include = <path relative to $HOME2L_ROOT/etc>
```

The available set of keys is tool-dependent. Unknown keys are ignored silently.

The set of commonly relevant parameters are listed in Section 4.7. Most tools have tool-specific parameters. They are listed with the documentation of the respective tool.

4.4.2 Section Specifiers

To allow assignments to be selectively active only for certain *Home2L* instances or on certain machines, sections can be defined:

```
[ <section specification> ]
```

The section specification defines to which instances the following parameter assignments apply. The section specification can be a single *tag* or an expression with multiple *tags* and logical operations.

The following *tags* are defined on an application startup.

- The machine's host name.
- The *Home2L* instance name (usually the name of the tool without the leading `home2l-`, user-defined scripts may define their own arbitrary instance names).
- The operating software environment (presently "Debian" or "Android").

i | The *Home2L host* ID is *not* available as a tag, since it may indirectly depend on settings of the `home2l.conf` file, and a cyclic dependency may occur.

Tags are case-sensitive. The user must take care that there are no name conflicts between machine names and instance names (both should be all lower-case). Section specifications may contain wildcards ('*'or'?'). In particular, the heading '`[*]`' starts a general section that applies to any instance.

The following logical operators are supported (ordered by decreasing precedence):

- Logical *NOT*: `'!'`
- Logical *AND*: `'&'`, `'@'`
- Logical *OR*: `','`, `'+'`

Examples:

```
[eniac,z3]      # Following settings apply to machines 'eniac' and 'z3'.

[wallclock@z3]  # Select only the Home2L WallClock instance on 'z3'.

[!shell]        # Select any instance except the Home2L Shell.

[*]             # Following settings apply to any instance.
```

4.5 Managing Background Services

The *Home2L Daemon* (`home2l-daemon`) can be used to start *Home2L*-related services at boot time, for example

- rules scripts,
- resource servers,
- doorman tasks,
- or any other user-specified shell script or command.

Services are specified by a `daemon.run.<script>` setting. They are kept alive by the daemon, crashed services are restarted automatically.

To enable the *Home2L Daemon* on a particular host, the file `$HOME2L_ROOT/install/initd-home2l` must be copied to `/etc/init.d/` as `home2l`. By default, the init script expects the *Home2L* installation blob to reside in `/opt/home2l`. To select a different path, create a file `/etc/default/home2l` with a line:

```
HOME2L_ENV=<your path to the 'env.sh' file>
```

The correct setting for a running installation can be obtained by:

```
home2l -e
```

As explained in Chapter 5, *Home2L resources* can be exported and delivered by any application linked to the *Home2l Resources* library. However, sometimes it is desired to run a process which does nothing but run some *Home2L drivers*. To this end, the tool `home2l-server` can be used. This tool does basically nothing, except for running the background services of the *Home2L* library and can thus be configured to run arbitrary drivers.

4.6 A Note on Security

Security is certainly serious aspect in networking and particularly smart home applications today. The security concept of the *Home2L* suite follows the Unix philosophy "Do one thing, and do this well" and the general philosophy to keep security-related things as simple and transparent as possible.

The *Home2Ls* do *not* implement any encryption or authentication mechanisms themselves, but are designed to rely on and colaborate with existing tools and mechanisms.

⚠ *Home2L* Security Rule

The *Home2L Resources* library and its networking operations assume to run on *trusted* computers in a *trusted* network. This is a requirement, and it is up to the user/administrator to provide such an environment, i. e. a trusted LAN with only trusted computers.

If there are untrusted machines in a private home (who wants that?), it is imperative to set up a trusted network for the *Home2L* cluster, e.g. using VLAN techniques or SSH tunnels.

The following configuration options are related to network security. This set is intentionally kept short and simple:

- `rc.enableServer`: If not set, *Home2L* will not listen on any networking port.
- `rc.serveInterface`: This option allows to restrict incoming connections to a certain physical network interface. If set to "local", only connections via the local interface (127.0.0.1) are accepted. This allows a setup where all peers are connected via secured SSH tunnels.
- `rc.network`: Declaration of the local network. Connection attempts from outside are dropped.

Violations against rules imposed by those settings such as connection attempts from outside the trusted network are logged with the special tag "SECURITY" to facilitate intrusion detection.

i The *Home2L* security rule is inspired and intended to correlate with how private buildings are usually protected in the real world: All parts of some well-definable *outer hull* (e.g. the outer walls) – the main door, windows, side entrances – are secured and lockable, wheras the interior is treated as a trusted area. Room doors inside a private home are usually left unlocked. (Or do you keep your kitchen door permanently locked as an additional protection against housebreaking?)

4.7 List of Common Configuration Parameters

4.7.1 Parameters of Domain `debug`

`debug` (int) [= 0] `common/base.C:44`

Level of debug output.

A value of 0 disables debug output messages. Higher values increase the verbosity of the debug output.

`debug.enableCoreDump` (bool) [= false] `common/env.C:48`

Enable generation of a core dump using the setrlimit() system call (without size limit).

4.7.2 Parameters of Domain `home2l`

`home2l.config` (string) [= "etc/home2l.conf"] `common/env.C:58`

Main configuration file (read-only).

`home2l.version` (string) `common/env.C:61`

Version of the Home2L suite (read-only).

`home2l.buildDate` (string) `common/env.C:64`

Build date of the Home2L suite (read-only).

`home2l.os` (string) [= ANDROID ? "Android" : BUILD_OS] `common/env.C:69`

Operation software environment (Debian / Android) (read-only).

This setting is determined by the build process.

`home2l.arch` (string) [= ANDROID ? NULL : BUILD_ARCH] `common/env.C:74`

Processor architecture (i386 / amd64 / armhf / <undefined>) (read-only).

This setting is generated during the build process and taken from the processor architecture reported by 'dpkg –print-architecture' in Debian.

4.7.3 Parameters of Domain `sys`

`sys.cmd.<name>` (string) [= NULL)] `common/base.C:2106`

Predefine a shell command.

For security reasons, shell commands executed remotely are never transferred over the network and then executed directly. Instead, a server can only execute commands predefined on the server side. This group of settings serves for pre-defining commands executed by a restricted shell.

`sys.syslog` (bool) [= false] `common/env.C:85`

Set to write all messages to syslog.

`sys.machineName` (string) `common/env.C:90`

System host name (read-only).

`sys.execPathName` (string) `common/env.C:93`

Full path name of the executable (read-only).

`sys.execName` (string) `common/env.C:96`

File name of the executable without path (read-only).

`sys.pid` (int) [= 0] `common/env.C:99`

System process ID (PID) (read-only).

`sys.instanceName` (string) `common/env.C:104`

Instance name (read-only).

The instance name should uniquely identify the running process. There is no technical mechanism to enforce uniqueness. Hence, it is up to the administrator take care of that.

The instance name can be set by the tool programmatically, or in some tools with the '-x' command line option. By default, the instance name is set to the name of the executable without an eventually leading "home2l-".

`sys.droidId` (string) [= "000"] `common/env.C:115`

Droid ID.

This is the 3-digit number displayed on the wall clocks to indicate the serial number of the device. If the host name ends with three digits, the droid ID is automatically taken from that.

`sys.rootDir` (string) `common/env.C:124`

Home2L installation root directory (HOME2L_ROOT).

`sys.varDir` (string) [= "/var/opt/home2l"] `common/env.C:127`

Root directory for variable data.

`sys.tmpDir` (string) [= "/tmp/home2l"] `common/env.C:130`

Root directory for temporary data.

`sys.locale` (string) `common/env.C:135`

Define the locale for end-user applications in the 'll_CC' format (e.g. "de_DE").

This setting defines the message language and formats of end-user applications. Only end user applications (presently WallClock) are translated, command line tools for administrators expect English language skills.

4.7.4 Parameters of Domain `phone`

`phone.ringbackFile` (path) [= "share/sounds/ringback.wav"] `common/phone.C:31`

Ringback file (Linphone backend only).

This is the sound to be played to the caller while ringing.

`phone.playFile` (path) `common/phone.C:37`

Phone play file (Linphone backend only).

This is the background music played to a caller during transfer. (may be removed in the future since PBX systems like ASTERISK already provide this functionality)

4.7.5 Parameters of Domain `daemon`

`daemon.minRunTime` (int) [= 3000] `tools/home2l-daemon.C:33`

Minimum run time below which a process is restarted only with a delay.

`daemon.retryWait` (int) [= 60000] `tools/home2l-daemon.C:36`

Restart wait time if a processes crashed quickly.

`daemon.pidFile` (string) `tools/home2l-daemon.C:39`

PID file for use with 'start-stop-daemon'.

`daemon.run.<script>` (string) `tools/home2l-daemon.C:42`

Define a script to be started and controlled by the daemon.

5 The *Resources* Library

5.1 Overview

A central part of the *Home2L* suite is the *Resources* library. It manages, provides access to and publishes what is referred to as *resources*. A ***resource*** can be anything that can provide or take data that may change over time. Examples for resources are:

- physical sensors (temperatur sensors, motion detectors, . . .),
- physical actors (window shades, room lights, . . .),
- computers (that may be woken up or shut down remotely),
- software services,
- run-time changable options for some software (for example, the *Home2L WallClock* can be requested to switch on/off or dimm the screen),
- run-time status reports of some software (for example, the *Home2L WallClock* can report the display brightness on Android tablets).

The resources are managed in a completely distributed way. There is no central server and, consequently, no single point of failure. Running program instances linked against the *Home2L Resources* communicate with each other on a peer-to-peer basis. Since they all share the same configuration, each peer is aware of the members of its cluster.

To deal with resources, operations are required to retrieve information (values and states) from sensing resources and to manipulate acting resources. However, there is a lot of parallelism and asynchronous behaviour in the system: Sensors (resources) may change their values any time (and sometimes very quickly), users behave asynchronously, and machines in a network behave asynchronously.

For reading out resources, a subscription and event model is implemented. If an application subscribes to a resource, each value or state change will be delivered instantaneously to the application by means of events (see Section 5.5). *Home2L* is very accurate here - all events (with new values or states) reported by a driver are delivered completely to any host subscribing to them. As an extreme example: If there is a mechanical switch which is not properly debounced, any host in the cluster is able to count the exact number of bounces!

As for the manipulation of resources, special care has to be taken with concurrency. For example, a user may push a button to open the window shades, and one second later, an automatic script may request the shades to close. What should happen now? *Home2L Resources* provides a resolution mechanism to deal with such situations in properly. Any instance which intends to manipulate a resource submits a *request* with the intended value and some parameters (e.g. time interval, priority), and the *Home2L Resources* library resolves them properly in the case of concurrency (see Section 5.6).

5.2 Concepts and Terminology

Figure 5.1 shows the terminology and interaction between three independent hosts. On each host, the application can manipulate any resource of the cluster by placing ***requests*** for value changes and read out resources by ***subscribing*** to them.

The *Home2L Resources* library takes care of

- request evaluation and resolution,
- event propagation to subscribers,
- network transparency and communication with other hosts,
- organizing all resources in a unified namespace (the directory).

Drivers to not have to worry about all that. Their task is to physically access their devices or whatever they are responsible for. Drivers can simply ***report*** new values from their devices, and, in the other direction, they receive ***drive*** API calls from the *Resources* library depending on the pending requests.

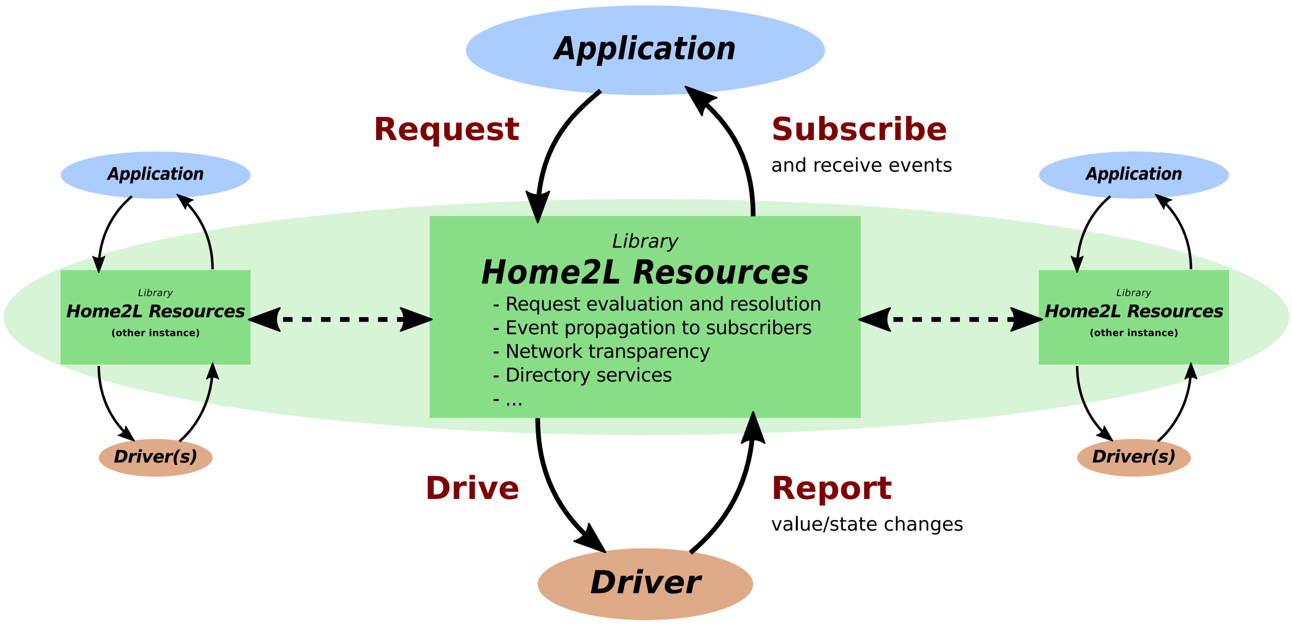


Figure 5.1: *Home2L* Resources: Terminology

Any application linked against the *Resources* library can provide drivers or load any of the existing drivers (see Chapter 7). This includes automation rules scripts (see Chapter 6), which may themselves implement drivers or execute external ones.

In order to "just" load driver(s) and export their resources to the cluster, the tool home2l-server can be used. This is basically an empty application without any functionality besides operate the *Resources* library.

5.3 Resources, Values and States

5.3.1 Resources

A *resource* object has the following attributes:

- a unique resource identifier (URI),
- a typed value and state,
- a "writeable" flag,
- a list of listening subscribers,
- a list of pending requests.

Resources are organized in a namespace resembling a directory tree with the following structure:

```
/host/                        ; Host domain: All resources, ordered by host/driver
    <host 1>/
        <driver 1.1>/
            <resource 1.1.1>  ; Resources may be arranged in sub-trees of arbitrary ...
            <resource 1.1.2>  ; ... depth by the driver.
            ...
        <driver 1.2>/
            <resource 1.2.1>
            ...
    <host 2>/
    ...
/alias/                       ; Alias domain: References to resources/subtrees in the ↩
    ↪host domain
    <alias 1>                 ; Aliases may be arranged in sub-trees and may point ..
    <alias 2>                 ; ... anywhere in the host domain (like symlinks).
    ...
/local/                       ; Alias for /host/<local host>/...
    <driver L.1>
        <resource L.1.2>
        ...
```

Except for the /local domain (a shortcut for a process to access its own resources), any host in the cluster has the same view on the resource tree.

5.3.2 Values

Resource values are typed, the following types are available:

a) Basic types

- `bool`: Boolean
- `int`: Integer
- `float`: Floating point
- `string`: Text string

b) Special types

- `trigger`: Triggerable events (example: `timer/daily`)
- `mutex`: Allows to implement mutual exclusion in a *Home2L* cluster (experimental)

c) Physical (or unit) types

- `percent ::= <float> %`
- `temp ::= <float> °C`

d) Enumeration types

- `window ::= { closed, open, tilted, openOrTilted }`

The set of unit types and enumeration types may be extended in the future. The following command prints an up-to-date list of all types for the current installation:

```
home2l shell -e types
```

5.3.3 States

A resource can assume one of the following states:

- `valid`: The value is known and valid.
- `busy`: The value is known, but the resource is busy.
- `unknown`: The value is unknown.

The *busy* state may indicate that an actor-like resource is anticipating some new value, but has not yet reached it. For example, if window shades are requested to close (value 0), the state/value may be *busy/0* (also written as " 0") while the shades are moving down and then change to *valid/0* when the shades are actually down. Or, if a computer is requested to start (value *true*), the driver may report a state/value of *busy/true* while the machine is booting and then later switch to *valid/true* as soon as it is ready to use.

The *unknown* state may be reported for a variety of reasons, for example:

- The driver may report it because its device does not deliver valid data.
- The network connection to the serving host was lost.
- The resource does not exist or the host is temporatily down.
- The resource is not subscribed to or not yet initialized.

It is important to note that application developers do not have to take extra actions for error checking. Whenever anything happens that may make the actual state unsure, the *Resources* library will turn the state to *unknown* and report this to the subscribers.

The state of a resource is not necessarily the same on all hosts. For example, a resource may be *valid* on the serving host, but *unknown* on some other host due to a network problem.

5.4 Configuring the Resources Library

The configuration file `resources.conf` defines hosts, aliases, and optionally signals for the *Home2L* cluster.

A *host* in this context is a *Home2L instance* running on a machine. In many cases, if there is only one instance serving resources, this instance can and should then be identified by its Unix host name in the etwork and listen on the default *Home2L* port (see below). If there are multiple serving instances running on the same machine, listening ports must be assigned manually for all but one of them.

The default port is defined by the following line:

```
P <port>
```

Hosts are declared as follows:

```
H <host ID> [ <hostspec> ]   ; <hostspec> ::= [<instance>@]<net host>[:<port>]
```

`<host ID>` is the host ID as visible by other instances. If no `<hostspec>` is given, `<host ID>` is equivalent to the network host name, and the default port is used.

`<net host>[:<port>]` is the network host name and port the server is listening on.

`<instance>` is the instance name of the respective server. It is only needed if multiple servers run on the same machine.

Examples:

```
H eniac   # Single host on machine 'eniac', using the default server port.
H z3-server    server@z3
          # The host ID 'z3-server' refers to 'home2l-server' on machine 'z3',
          # and the server listens on the default port.
H z3-doorman   wallclock@z3:4701
          # The host ID 'z3-doorman' refers to 'home2l-doorman' on machine 'z3',
          # and the server listens on port 4701..
```

Servers/hosts are also implicitly declared by the aliases and variables, which is equivalent to an explicit declaration without `<hostspec>`.

Aliases are defined with the following syntax:

```
A <aliasName> <destPath>
```

`<destPath>` may be relativ to `/host/` or absolute. Examples:

```
A tempOutside          /host/gatewayhost/weather/tempOutside
A door/backlight       sphinx/gpio/doorBacklight
```

These make the current outside temperature (presently delivered via internet by host `gatewayhost`) accessible as `/alias/tempOutside` and the backlight of the door button as `/alias/door/backlight`.

Signals are resources that do not drive any hardware or software, but simply report back all values driven to them (similar to loopback devices in Unix). They are useful for intermediate values (for example, the logical "and" of a motion sensor and darkness sensor to indicate whether a light should be switched on) or for testing purposes (for example, to temporarily replace a real device by the signal resource, which can be manipulated manually). They are defined as follows:

```
S <host> <name> <type> [<default value>]
```

Signals are handled by the `home2l-drv-signal` driver, all signals are read- and writable. Their *URI* is `/host/<host>/signal/<name>`. Examples:

```
S turing testBool  bool
S turing testInt   int   7
S turing testFloat float 3.14
```

Checklist: Enabling a *Home2L* Server

To enable a local server, the following conditions must be met:

1. The application must enable it when calling 'RcInit()' (C/C++) or 'Home2lInit()' (Python).
2. The configuration option `rc.enableServer` must be 'true'.
3. It must be declared as a host in `resources.conf`.

Client functionality is always available.

5.5 Subscriptions

Subscriptions are handled by the class `CRcSubscriber` (see C/C++ API). A subscriber object can subscribe to any number of resources, eventually selected by wildcards. Resources do not need to exist yet at the time they are subscribed to.

Subscriber events can be received in multiple ways (see class `CRcEventProcessor`):

- asynchronously by providing a callback function,
- synchronously and blocking (`CRcEventProcessor::WaitEvent ()`),
- synchronously and non-blocking (`CRcEventProcessor::PollEvent ()`).

The following types of events may be delivered (see class `CRcEvent`):

- `rceValueStateChanged`: The resource has changed its value.
- `rceDisconnected`: The connection to the (remote) resource has been lost.
- `rceConnected`: The connection to the (remote) resource has been (re-)established.

The events *rceDisconnected* and *rceConnected* are relevant, if it is important to capture each value/state change and not just to get the most up-to-date value. These events indicate whether a gapless event sequence is guaranteed or not.

For those who are missing "just a normal read" operation: As long as a resource is subscribed to by some subsciber, a call to `CResource::GetValueState()` and its relatives will always return the most up-to-date value and state.

With the Python API, subscribers are used internally if the decorators `@onEvent(<resources>)` or `@onUpdate(<resources>)` are used. Functions decorated with the former are executed with each event on one of its resources. The latter is more efficient since it drops value/state changes if they are already outdated at the time they are fetched.

5.6 Requests

In order to properly handle concurrent manipulating accesses to resources, a *request resolution* mechanism is implemented, which loosely resembles the concept of *resolution functions* in hardware description languages (VHDL, Verilog and friends). Applications never "write" to a resource (this would cause serious concurrency issues), they place *requests*.

Each *request* (class `CRcRequest`) is associated with a **resource**, **a value** (no state) and a *request identifier (ID)*. The ***request ID*** allows to later modify or delete the request from any host.

Additionally, a request may have the following optional attributes:

Priority (Syntax: `*<prio>`): The priority of the request.

This is a number between 0 (= lowest) and 15 (= highest). The default priority is 8. If multiple requests at the same priority exist, the oldest one dominates. (This behavior is required to allow the implementation of Mutexes, where lock owners cannot be preempted.)

***Start time* (or *on time*)** (Syntax: `+<time>`):

The request becomes active then, but is not considered before.

***End time* (or *off time*)** (Syntax: `-<time>`):

The request will expire and be discarded after this time.

Hysteresis (Syntax: `~<hyst>`): Hysteresis time.

> This allows the *Resources* library to postpone the requested action by up to `<hyst>` milliseconds in order to avoid switches forth and back again (for example, shutting down a computer that is needed again soon).
>
> If a positive hysteresis is given, a request will only be activated if no change to a different value is planned (according to existing requests) within the hysteresis time. Only follow-up events without a hysteresis or a hysteresis small enough that they cannot be pushed out of the current hysteresis interval are considered.

If no requests are active for a resource, its value will be left unchanged. It is legal to set the start and end times to identical values, in which case the value is set once at the specified time (given that no other request with a higher priority prohibits that) and the request is deleted again.

Resources of type `trigger` are processed specially: Only at the start time, trigger events are generated. Hence, it is recommended to supply an end time (which may be equal to the start time) to auto-remove the request.

i

Examples

1. The user pushes a button to open the window shades. One second later, an automatic script decides that the shades should be closed. What should happen now? An answer to this question can only be given by the end user. However, the *Home2L's* request mechanism allows to implement various different solutions.

   Possible solution A:

   - The automatic script sets permanent requests `#script` for a certain value (e. g. 0 for "up" and 1 for "down") and a priority of 5 (the default priority).
   - Pushing one of the user buttons "up" and "down" generates a timed requests `#manual` with the off time attribute set to some time in the future (e. g. 1 hour after the button was pushed) and an increased priority of 6. This request will dominate over the script's request, and after the off time, the script request will take over again.

   Possible solution B:

   - Same as solution A, but with the requests generated with the "up" and "down" button being permanent.
   - A third push button is used with a "stop manual mode" functionality. Pushing it removes the `#manual` request.

2. A PC-based video recorder can be requested to be on by recording timers as well as manually by the user to watch TV. Whenever unused, the PC should shut down automatically, but only if the time of the next recording is more than 30 minutes in the future to avoid unnecessary power cycles.

   This can be modelled as follows:

   - The video recorder sets timed requests `#timer` (with appropriate start and end times) for its recordings with a boolean value of '1'.
   - The user sets and deletes untimed manual requests `#namual` with a value of '1'.
   - To let the computer shut down if not needed, a permanent request `#default` with a low priority and a value of '0' is set with a hysteresis of 30 minutes.

5.7 Syntax of Value and Request Specifications

In some places, particularly in the `home2l-shell` and the Python API, values, states or requests are printed or accepted in textual form. This section explains the syntax commonly used for value/state objects and requests.

The syntax of a value/state object is:

```
<valueState> ::= [ "(" <type> ")" ] ( [!]<value>|? ) [ @<time> ]

<value> ::= <bool> | <int> | <float> | <string> | <time> | <unitval> | <enumval>
```

```
  <bool>    ::= [0fF] | [1tT+]               : Boolean value
  <int>     ::= [-][0-9]+                    : Integer value
  <float>   ::= [-][0-9]*[.[0-9]+][E[+/-][0-9]+]  : Floating-point value
  <string>  ::= [0-9a-zA-Z\]+ | \0           : String (\-escaped, UTF8 encoding)
  <time>    ::= <time>                       : Time value
  <unitval> ::= [<float>|<int>]<unit>        : Unit value (<unit> is the unit string)
  <enumval> ::= [_a-zA-Z][_a-zA-Z0-9]+       : Enumeration value
```

A `<value>` must not contain any spaces, type and timestamp attributes are separated by spaces.

Requests are specified as follows:

```
<request> ::= <value> [ <attributes> ]

  <attributes>' is a space-separated subset of:

    #<id>   : Request ID [default: instance name]
    *<prio> : Priority (0..9) [default: 5]
    +<time> : Start time
    -<time> : End time
    ~<hyst> : Hysteresis in milliseconds
```

Time values – either as a value or a an start/end time specification – can be specified in several alternative formats:

```
YYYY-MM-DD[-hhmm[ss[.<millis>]]] : Date and time, interpreted as local time

t<unsigned integer>          : Absolute time in milliseconds since the Epoch (POSIX time)

<integer>                    : Relative time, milliseconds from now

hh:mm[:ss[.<millis>]]        : Time relative to 0:00 today; 'hh' may be > 23
                               to specify a time in the coming days
```

With the evolution of the *Home2L* software, the information above may become outdated or incomplete. The most up-to-date information can be found in the following places of the C/C++ API documentation:

- `CRcValueState::SetFromStr()` (source file: `resources/resources.H`)
- `CRcRequest::SetFromStr()` (source file: `resources/resources.H`)
- `TicksFromString()` (source file: `common/base.H`)

5.8 The *Shell* – The Command Line Interface to *Resources*

The *Home2L Shell* is a command line interface to the *Resources* library and can be seen as a "swiss army knife" to access and inspect the resources and servers of a *Home2L* cluster.

With the *Home2L Shell* you can:

- list all (server) hosts in the cluster and check their status and availability,
- list and inspect the resources directory,
- for each resource, see its current value and state, its current list of subscribers and all currently active requests,
- manually set and delete requests,
- monitor resource events,
- load and run drivers,
- get information on *Home2L* itself (e.g. the list of available value types).

The *Home2L Shell* can be run interactively or execute commands in batch mode. The latter can be used to set/delete requests from a shell script and for logging of measurements.

Details on the usage of the *Home2L Shell* can be obtained by the 'help' command:

```
$ home2l shell
home2l> help
```

Details on the batch usage can be obtained by:

```
$ home2l shell -h
```

Section 2.6 contains detailed examples for working with the *Home2L Shell*.

5.9 Integrated Drivers

5.9.1 Driver *Signal*

The *Signal* driver is always available and allows to declare resources which just report back any driven value without any technical functionality.

Signals can serve as intermediate resources or for testing purposes.

They can be defined inside a `resources.conf` configuration file (see Section 5.4) or by the API calls `RcRegisterSignal()` (C/C++) or `NewSignal()` (Python).

5.9.2 Driver *Timer*

The *Timer* driver provides resources reflecting the current time, triggers to initiate hourly or daily tasks, and a set of resources reflecting day and night times, suitable, for example, to control an automatic outdoor light.

The driver is statically built into the *Resources* library. It can be disabled using the `rc.timer` setting.

The *Timer* driver exports the following resources:

`timer/twilight/day` (bool,ro) `resources/rc_drivers.C:91`

Flag to indicate day time (time between official sunset and sunrise).

`timer/twilight/day06` (bool,ro) `resources/rc_drivers.C:94`

Flag to indicate civil day time (time between civil dawn and dusk).

`timer/twilight/day12` (bool,ro) `resources/rc_drivers.C:97`

Flag to indicate nautical day time (time between nautical dawn and dusk).

`timer/twilight/day18` (bool,ro) `resources/rc_drivers.C:100`

Flag to indicate astronomical day time (time between astronomical dawn and dusk).

`timer/twilight/sunrise` (time,ro) `resources/rc_drivers.C:104`

Today's official sunrise time.

`timer/twilight/dawn06` (time,ro) `resources/rc_drivers.C:107`

Today's civil dawn time.

`timer/twilight/dawn12` (time,ro) `resources/rc_drivers.C:110`

Today's nautical dawn time.

`timer/twilight/dawn18` (time,ro) `resources/rc_drivers.C:113`

Today's astronomical dawn time.

`timer/twilight/sunset` (time,ro) `resources/rc_drivers.C:117`

Today's official sunset time.

`timer/twilight/dusk06` (time,ro) `resources/rc_drivers.C:120`

Today's civil dusk time.

`timer/twilight/dusk12` (time,ro) `resources/rc_drivers.C:123`

Today's nautical dusk time.

`timer/twilight/dusk18` (time,ro) `resources/rc_drivers.C:126`

Today's astronomical dusk time.

`timer/now` (time,ro) `resources/rc_drivers.C:268`

Current time (updated once per second).

`timer/daily` (trigger,ro) `resources/rc_drivers.C:272`

Triggers once per day (shortly after midnight).

`timer/hourly` (trigger,ro) `resources/rc_drivers.C:275`

Triggers once per hour (at full hour).

`timer/minutely` (trigger,ro) `resources/rc_drivers.C:278`

Triggers once per minute (at full minute).

5.10 List of Configuration Parameters

5.10.1 Parameters of Domain `rc`

`rc.config` (string) [= "etc/resources.conf"] `resources/rc_core.C:76`

Name of the Resources configuration file.

`rc.enableServer` (bool) [= false] `resources/rc_core.C:80`

Enable the Resources server.

(Only) if true, the Resources server is started, and the local resources are exported over the network.

`rc.serveInterface` (string) [= "any"] `resources/rc_core.C:88`

Select interface(s) for the server to listen on.

If set to "any", connections from any network interface are accepted.

If set to "local", only connection attempts via the local interface (127.0.0.1) are accepted. This may be useful for untrusted physical networks, where actual connections are implemented e.g. by SSH tunnels.

If a 4-byte IP4 address is given, only connections from the interface associated with this IP address are accepted. This way, a certain interface can be selected.

This value is passed to bind(2), see ip(7) for more details. The value of "any" corresponds to INADDR_ANY, the value of "local" corresponds to INADDR_LOOPBACK.

`rc.network` (string) [= "127.0.0.1/32"] `resources/rc_core.C:105`

Network prefix and mask for the Resources server (CIDR notation).

Only connections from hosts of this subnet or from 127.0.0.1 (localhost) are accepted by the server.

`rc.resolve.<alias>` (string) `resources/rc_core.C:112`

Define a manual network host resolution.

When a network host is contacted, this environment setting is consulted by the client before any system-wide name resolution is started. This can be used, for example, with SSH tunnels to map the real target host to something like 'localhost:1234'.

Another use case is just a hostname resolution independent of a DNS service or '/etc/hosts' file, which can be useful on Android client devices, for example.

`rc.maxAge` (int) [= 60000] `resources/rc_core.C:123`

Maximum age (ms) tolerated for resource values and states.

If a client does not receive any sign of life from a server for this amount of time, the resource is set to state "unknown" locally. Servers send out regular "hello" messages every 2/3 of this time. Reducing the value can guarantee to detect network failures earlier but will increase the traffic overhead for the "hello" messages.

This value must be consistent for the complete Home2L cluster.

`rc.netTimeout` (int) [= 3000] `resources/rc_core.C:135`

Network operation timeout (ms).

Waiting time until a primitive network operation (e.g. connection establishment, response to a request) is assumed to have failed if no reply has been received.

`rc.netRetryDelay` (int) [= 60000] `resources/rc_core.C:142`

Time (ms) after which a failed network operation is repeated.

Only in the first period of `rc.netRetryDelay` milliseconds, the connection retries are performed at faster intervals of `rc.netTimeout` ms.

`rc.netIdleTimeout` (int) [= 5000] `resources/rc_core.C:148`

Time (ms) after which an unused connection is disconnected.

`rc.relTimeThreshold` (int) [= 60000] `resources/rc_core.C:152`

Threshold (in ms from now) below which remote requests are sent with relative times..

This option allows to compensate negative clock skewing effects between different hosts. If timed requests are sent to remote hosts, and the on/off times are in the future and in less then this number of milliseconds from now, the times are transmitted relative to the current time. This way, the duration of requests is retained, even if the clocks of the local and the remote host diverge. (Example: A door opener request is timed for 1 second and should last exactly this time.)

`rc.maxOrphaned` (int) [= 1024] `resources/resources.C:743`

Maximum number of allowed unregistered resources.

Resource objects (`class CResource`) are allocated on demand and are usually never removed from memory, so that pointers to them can be used as unique IDs during the lifetime of a programm. Unregistered resources are those that presently cannot be linked to real local or remote resource. They occur naturally, for example, if the network connection to a remote host is not yet available. However, if the number of unregistered resources exceeds a certain high number, there is probably a bug in the application which may as a negative side-effect cause high CPU and network loads.

This setting limits the number of unregistered resources. If the number is exceeded, the application is terminated.

`rc.timer` (bool) [= true] `resources/rc_drivers.C:55`

Enable/disable the 'timer' driver.

`rc.drvMinRunTime` (int) [= 3000] `resources/rc_drivers.C:349`

Minimum run time of a properly configured external driver (ms).

To avoid endless busy loops caused by drivers crashing repeatedly on their startup (e.g. due to misconfiguration), a driver crashed on startup is not restarted immediately again, but only after some delay.

This is the time after which a crash is not handled as a startup crash.

`rc.drvCrashWait` (int) [= 60000] `resources/rc_drivers.C:358`

Waiting time (ms) after a startup crash before restarting an external driver.

To avoid endless busy loops caused by drivers crashing repeatedly on their startup (e.g. due to misconfiguration), a driver crashed on startup is not restarted immediately again, but only after some delay.

This parameter specifies the waitung time.

`rc.drvMaxReportTime` (int) [= 5000] `resources/rc_drivers.C:367`

Maximum time (ms) to wait until all external drivers have reported their resources.

`rc.drvIterateWait` (int) [= 1000] `resources/rc_drivers.C:370`

Iteration interval (ms) for the manager of external drivers.

5.10.2 Parameters of Domain `drv`

`drv.<id>` (string) `resources/rc_drivers.C:327`

Declare/load an external (binary or script-based) driver.

The argument <arg> may be one out of:

a) The name of a driver .so file (binary driver).

a) The invocation of a script, including arguments.

a) A '1', in which case <id> is used as <arg> (shortcut to enable binary drivers).

Relative paths <name> are search in:

- <HOME2L_ROOT>/etc[/<ARCH>]
- <HOME2L_ROOT>/lib/<ARCH>/home2l-drv-<name>.so
- <HOME2L_ROOT>/lib/<ARCH>/home2l-drv-<name>
- <HOME2L_ROOT>/lib[/<ARCH>]
- <HOME2L_ROOT>/

Please refer to the section on writing external drivers in for further information on script-based drivers.

5.10.3 Parameters of Domain `shell`

`shell.historyFile` (string) [= ".home2l_history"] `resources/home2l-shell.C:36`

Name of the history file for the home2l shell, relative to the user's home directory.

`shell.historyLines` (int) [= 64] `resources/home2l-shell.C:39`

Maximum number of lines to be stored in the history file.

If set to 0, no history file is written or read.

`shell.stringChars` (int) [= 64] `resources/home2l-shell.C:44`

Maximum number of characters to print for a string..

If set to 0, strings are never abbreviated.

5.10.4 Parameters of Domain `location`

`location.latitudeN` (float) [= 48.371667] `resources/rc_drivers.C:64`

WGS84 coordinate (latitude north) of the building.

This value is (amoung others) used by the 'timer' driver for twilight calculations.

`location.longitudeE` (float) [= 10.898333] `resources/rc_drivers.C:70`

WGS84 coordinate (longitude east) of the building.

This value is (amoung others) used by the 'timer' driver for twilight calculations.

6 Writing Automation Rules

6.1 Overview

With the *Home2L* suite, automation scripts are typically written in Python. Automation scripts are normal Python programs. They can run on any machine, there can be multiple of them, and they can be started or stopped any time. The latter is particularly useful for testing and debugging. Even interactive work in a Python shell is possible.

For a quick start, it is recommended to read the commented sample rules file also used in the tutorial (Chapter 2). Also, it is worth looking into the implementation of the tutorial's *ShowHouse*, which technically is implemented in Python like a rules script (see Sections 2.9 and 2.10.

Typically, a rules script performes a number of initializations, declares a number of triggered functions, and finally calls `Home2lRun()`) to enter the main event loop of the *Home2L* library/package. While the native *Home2L* library makes use of multi-threading, all Python functions are (and must be) generally called from the main thread, so that no synchronization measures have to be implemented in Python code.

The following subsections give some additional hints and references to the respective places in the Python API documentation, which serves as a reference manual for rules writing.

6.2 Triggered Functions

6.2.1 Resource Events and Value/State Updates

Functions to be run can at certain (resource) events can be declared with the help of the `@onEvent()` decorator:

```
@onEvent ( <resources> )
def MyFunc (ev, rc, vs):
  # user code
```

Functions decorated this way are called for each individual event. This allows to receive all events without simplificatiopns in the correct order (important to not miss any event or quick value change) as well as "connected" and "disconnected" events.

As arguments, the decorator expects a set of resources, which may be a single resource or a tuple or list, and each resource may be specified by a string with its URI or by a reference to the resource object previously retrieved by `RcGetResource()`.

The argument `ev` passed to the user function identifies the type of event, which can:

- `rceValueStateChanged:` The value or state has changed.
- `rceConnected:` The connection has been (re-)established.
- `rceDisconnected:` The connection has been lost.

The arguments `rc` and `vs` are the resource and their current value and state (see class `CRcValueState`), respectively.

i To avoid race conditions and maintain the precise event model, it is important to never query the passed `rc` for its value/state, but to use the passed `vs` instead.

6.2.2 Value/State Updates

Functions decorated with `@onUpdate()` are called each time the value or state of a resource changes:

```
@onUpdate ( <resources> )
def MyFunc ():
  # user code
```

If the value or state changes very quickly, the user function may be called only for the latest, most up-to-date value/state. Unlike functions decorated with `@onEvent()`, events may automatically be dropped to optimize performance. This decorator is recommended if it is sufficient to always have the most up-to-date value/state of a resource, whereas the precise sequence of events is not relevant. If it is important to not miss any event or quick value change, `@onEvent()` should be used.

i `MyFunc()` has no arguments here. Unlike the precise event model, with the "always up to date" model assumed here, it is not critical to read the value/state of some resource inside the user function. The worst thing that can happen is that a value newer than the value that caused the invocation of the user function is retrieved.

6.2.3 Timed Functions

To run a function a certain time or at certain intervals, the following decorator can be used:

```
@at ( t = <t>, dt = <interval> )
def MyFunc ():
  # user code
```

The `dt` argument is optional. If unset, the function is called once only.

Sometimes, a timed function is to be run multiple times in different situations. The reuse of the user function (`MyFunc()`) is simplified, if

```
RcRunAt (func, t = 0, dt = 0, data = None)
```

is used instead of a decorator.

6.2.4 Daily Requests

Often, static requests need to be set that are re-calculated on a daily basis. For example, an outdoor light might need to be switched on at night depending on the current sunset and sunrise times. In this case, a *daily rule* my be defined, which calculates the light-on times at midnight and then sets a request for the light resource accordingly.

The following decorator defines a function for such daily requests.

```
@daily ( <host set> )
def MyFunc (host):
  ...
```

As an argument, the decorator expects a set of host IDs, which is a list or tuple of strings. The host ID list is used to monitor the availability of the hosts and eventually also run the user function after a host has crashed or being restarted and becomes reachable (again). Besides that, the user function `MyFunc()` is called with the host ID as an argument on each day, shortly after midnight. It is called once per host, so that it should only set requests for the host identified by the argument.

6.3 Retrieving Resource Values and States

Due to the highly concurrent nature and implementation of the *Resources* library, resource values and states may change any time. To securely retrieve values und states without race conditions, they must first be captured in a `CRcValueState` object:

```
vs = rc.ValueState()
```

The `CRcValueState` object `vs` now contains information about the type, the value, and the state of the resource.

i To functions decorated with `@onEvent()`, an appropriately captured value/state object is already passed by the `vs` argument, and that must be used in order to avoid race conditions.

The value to can then be accessed by:

```
val = vs.Value ()
```

If the state of the `vs` object is 'unknown', `Value()` returns `None`. The caller must consider this and the code must consistently be written in a way that all calls of the `Value()` method can return `None` any time.

To simplify rules development, the method

```
val = vs.ValidValue(default)
```

can be used instead, which alway returns a valid (non-None) value. If the actual value is invalid, the passed `default` value is returned.

The following methods may be useful to obtain additional attributes (refer to the Python API or C/C++ API documentation for details):

```
Type ()      # the type
State ()     # the state
TimeStamp () # the time stamp
IsValid ()   # True, if the state is 'rcsValid', neither 'rcsBusy' nor 'rcsUnknown'.
IsKnown ()   # True, if the value can be retrieved (state is 'rcsValid' or 'rcsBusy')
```

6.4 Placing requests

Requests can be set or deleted by the following functions:

```
RcSetRequest (resource, reqGid, <args>)
RcSetRequestFromStr (resource, reqDef)
RcDelRequest (resource, reqGid)
```

The `resource` argument can either be a string with the URI or a reference to the resource object previously retrieved by `RcGetResource()`. The `reqGid` parameter is the request ID (a string value).

The additional arguments (`<args>`) contain the value and optional attributes of the request.

The `RcSetRequestFromStr()` variant allows to pass all request attributes by a single string with the same syntax as supported by `home2l-shell` and specified in Section 5.7.

7 Writing Drivers

7.1 Binary Drivers

Examples: *Demo (`drivers/demo`), GPIO (`drivers/gpio`)*

Binary drivers are written in native C/C++ code and compiled to a shared object (`.so`) file, which at run time can be loaded by a `drv.<id>` configuration entry.

The driver must contain a single entry function declared as follows:

```
HOME2L_DRIVER(<name>)
(ERcDriverOperation op, CRcDriver *drv, CResource *rc, CRcValueState *vs) {
  switch (op) {
    case rcdOpInit:
      ...
      break;

    case rcdOpStop:
      ...
      break;

    case rcdOpDriveValue:
      ...
      break;
  }
}
```

As outlined here, the driver must provide up to three operations:

The "Init" operation registers all resources and initializes the driver itself.

The "Stop" operation is called just before the driver gets unloaded. It must stop/join all own background threads and shut down its own operations. Resources (`CResource` objects) do *not* have to be unregistered, this will be done automatically later.

The "DriverValue" operation is only called and required for writable resources. It must drive a new value to the real device (e.g. some actor) to make something happen. It is *neither necessary nor allowed* to call any `CResource` method here. The sole task of the driver is to operate its hardware, the driven value with a state "valid" will be reported automatically.

If it is not appropriate to report the passed value and a "valid" state back this time (e.g. due to an error or if the actor requires some time to fulfill the requested action), the passed `vs` object should be modified accordingly.

> *i* For example, a driver for window shades which is requested to close the shades would now modify the `vs` object and set its state to "busy" to indicate that the shades are now moving down. Later, when they are actually closed, the driver calls `CResource::ReportValue()` with a value/state of "1/valid" (assuming the value for closed shades is "1") to report the action was completed successfully.

The reporting of values for sensor-like hardware is typically done asynchronously. The driver will probably start its own background thread during initialization which communicates with the hardware. New values for some resource can be reported any time from any thread by simply calling one of the `CResource::ReportValue()` methods (see the C/C++ API for details).

7.2 Script-Based Drivers

Example: *Weather (`drivers/weather`)*

A *script-based* driver is typically implemented as a shell script and communicates with the *Resources* library by its standard input and output by means of simple text lines. To get an impression of the simple protocol, one may call such a driver on the command line, for example:

```
$(HOME2L_ROOT)/lib/home2l-drv-weather
```

At the script developer's choice, the driver can be operated in one of two modes:

a) *Keep-running mode:* The script is running permanently (and eventually restarted automatically if it crashes).

b) *Polling mode:* The script is run for dedicated operations and expected to terminate as soon as these operations are completed.

The script is called by the *Resources* library with the following arguments:

```
<script> -init     # Initialize driver, driver must report its properties (see below)

<script> -restart  # Restart driver (only after abnormal stop in keep-running mode);
                   # The driver does not need to report anything.

<script> -poll     # Driver is polled for new readable values
                   # (only in polling mode)

<script> -drive <resource LID> <value>
                   # Drive a value (only in polling mode);
                   # The driver must report the result by "v" messages.
```

In "keep-running" mode, values to drive are passed to the script's standard input as lines with the following format:

```
<resource LID> <value/state>
```

The resource's *local ID (LID)* is part of the *URI* following the driver name. During initialization (`'-init'` call), the driver is expected to output a series of lines with the following contents:

```
d <resource LID> <options>  # Declare a resource
p <poll interval>           # Define the polling interval
                            # (0 = no polling; Default = no polling)

.                           # Initialization complete - enter polling mode
:                           # initialization complete - enter "keep running" mode
```

The initialization phase must be completed as quickly as possible in the beginning.

In the active phase, lines with the following contents can be written:

```
v <resource LID> <value/state>  # Report a new value/state.
p <poll interval>               # Change the polling interval (polling mode only).
```

7.3 Python Drivers

Example: *The ShowHouse (`resources/home2l-showhouse`)*

As a third option, drivers can be defined as part of a Python script.

i The demo home2l-showhouse implements a driver for a) a set of (simulated) sensors and actors for a virtual building, but also b) an asynchronous keyboard driver for its own UI operation. The latter demonstrates how to work with background threads.

A driver is defined using the decorator

```
@newDriver ( <driver name> )
def DriverFunc (rc, vs):
  ...
```

and resources for it are registered by the function

```
RcNewResource (driverName, resourceLID, rcType, writable)
```

To improve code readability, `RcNewResource()` invokations are allowed before the driver is registered, in which case the resources are registered later together with the driver. Both drivers and their resources *must* be registered during the elaboration phase before calling `Home2lStart()` or `Home2lRun()`.

The driver function `DriverFunc()` is responsible for driving values to writable resources and is the equivalent to the `rcdOpDriveValue` operation in a binary driver (i.e. must not do anything else than manipulating its hardware and may eventually report a failure by changing the state or value of the passed `vs` reference.

Value changes (for sensing resources) can be reported by the `CResource::ReportValue()` method. With the Python API, this is the only method that can be called from any thread. All other API functions must be called from the main thread.

8 Supplied Drivers

8.1 Driver *GPIO*

8.1.1 Description

The *GPIO* driver is a universal driver leveraging the Linux `sysfs` GPIO capabilities to access general purpose inputs and outputs (GPIO).

In order to allow GPIOs to be used from a normal user application, they must be set up properly beforehand.

This preparation requires `root` privileges and is therefore done by the *Home2L* init script at boot time. The names and configurations (e. g. direction, initial value) of available GPIOs are defined by symbolic links residing in

```
$HOME2L_ROOT/etc/gpio.<machine name>
```

pointing to the actual device, typically:

```
/sys/class/gpio/gpio<n>
```

The links are read both by the *GPIO* driver and the init script, and they must conform to the following naming conventions:

```
$HOME2L_ROOT/etc/gpio.<machine name>/<port name>.<options>

<options> is a sequence of characters and may include:
    i - The port is an input.
    0 - The port is an output with a default value of 0.
    1 - The port is an output with a default value of 1.
    n - The port is active-low (negated).
```

8.1.2 Exported Resources

The exported resources depend on the configuration (see above). They are named after their port names (`gpio/<port name>`) as specified in the name of the symbolic link.

8.2 Driver *Weather*

8.2.1 Description

The *Weather* driver provides local weather information by querying the Open Data service of the German Weather Service (DWD).

⚠ Other services (e. g. for weather outside of Germany) are presently *not* supported.

8.2.2 Configuration Parameters

`weather.stationID` (int) [= NULL)] `drivers/weather/home2l-drv-weather:57`

Station ID according to the DWD station table.

See "MOSMIX-Stationskatalog", available for download at https://www.dwd.de/opendata .

`weather.debug` (int) [= False)] `drivers/weather/home2l-drv-weather:67`

Run the driver in debug mode.

In debug mode, radar eye images are not exported as a resource, but written into a local image file. This facilitates to run the driver directly on the command line for debugging purposes.

8.2.3 Exported Resources

`weather/temp` (temp,ro) `drivers/weather/home2l-drv-weather:106`

Outside temperature.

`weather/pressure` (float,ro) `drivers/weather/home2l-drv-weather:109`

Outside air presssure reduced to mean sea level.

`weather/humidity` (percent,ro) `drivers/weather/home2l-drv-weather:112`

Relative humidity.

`weather/windDir` (int,ro) `drivers/weather/home2l-drv-weather:115`

Mean wind direction during last 10 min. at 10 meters above ground.

`weather/windSpeed` (float,ro) `drivers/weather/home2l-drv-weather:118`

Mean wind speed during last 10 min at 10 meters above ground.

`weather/windMax` (float,ro) `drivers/weather/home2l-drv-weather:121`

Maximum wind speed last hour.

`weather/weather` (int,ro) `drivers/weather/home2l-drv-weather:124`

Present weather condition according to a table provided by the DWD..

See file 'poi_present_weather_zuordnung.pdf', available for download at https://www.dwd.de/opendata .

`weather/radarEye` (string,ro) `drivers/weather/home2l-drv-weather:225`

Radar Eye.

An .pgm-encoded image of 128x128 pixels showing the weather radar around the own position configured by as configured by the 'location.*' settings together with the wind direction and strength.

The cross indicates the ego position and is shifted away from the center such that the wheather indicated in the center of the radar eye will reach the own position in 2 hours. The cross has a radius of 50 km.

`weather/radarWarning` (bool,ro) `drivers/weather/home2l-drv-weather:237`

Radar Warning (NOT IMPLEMENTED YET).

Flag indicating that it may be raining and any roof windows should be closed.

9 *WallClock* – A Nonobtrusive GUI

9.1 Overview

The *WallClock* intends to appear as what the name suggests – a wall clock, but with some extra functionality in an unobtrusive way.

The main display (see Figure 9.1) shows the time and date, local weather information (if the `home2l-drv-weather` driver is set up) and some status about the building (see Section 2.5.2 for more explanations).

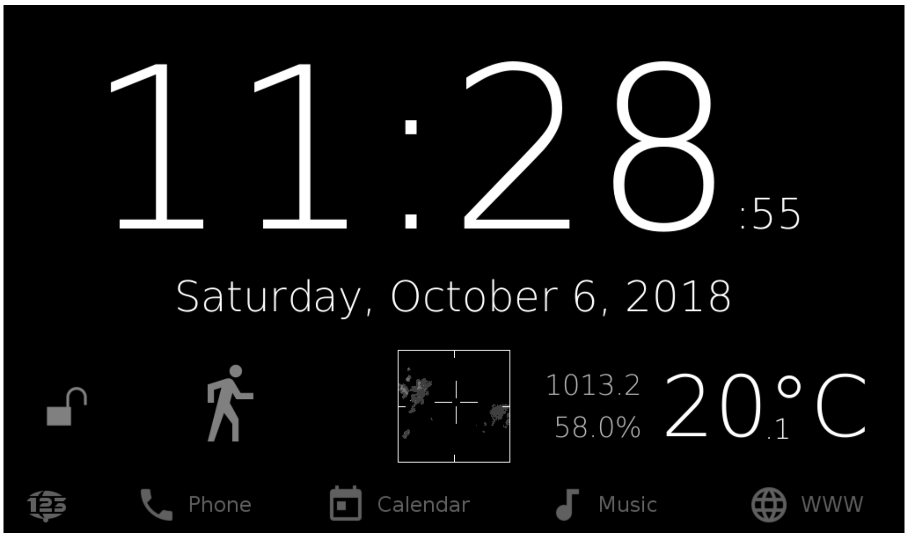


Figure 9.1: The *WallClock* Main Screen

The following functional modules (referred to as *applets*), which are frequently useful in a private household, are integrated in the *WallClock*:

- a SIP-based video phone client (for intercommunication and doorphone funtionality),
- a calendar,
- a music player client (*MPD* client) designed for a distributed setup with many rooms and muktiple user in a household,
- an alarm clock to use the *WallClock* device as a radio alarm clock.

Ideally, a *WallClock* device is installed in each room in the house. Due to its resource-aware implementation, low-cost hardware such as cheap or older second-hand Android tablets should be sufficient.

The *WallClock* is implemented in native code (C/C++) and uses *SDL2* for its UI. This makes it portable and efficient. So far, the *WallClock* has been tested on (PC) Linux and on Android, but ports of *SDL2* to many other environments exist (see `libsdl.org`).

i The visualization of the door lock(s) and motion sensor in the lower left of the home screen are just a temporary. Future versions will contain a generalized solution based on a zoomable floorplan, which will be capable of presenting the status of arbitrary gadgets here.

9.2 The Phone

The *Phone* applet implements a SIP-based VoIP phone with video fumctionality and the ability to act as a door phone in conjunction with `home2l-doorman`. For example, the applet can offer a door opener button if its peer is a door phone.

Together with a private branch exchange (PBX) software such as *Asterisk*, the *WallClocks* can be used as an intercom system for the house and for a sophisticated door phone system with multiple door bells and multiple answering stations in the building.

In order to compile the *WallClock* with phone capabilities, *PJSIP* or *libLinphone* is required.

9.3 The Calendar

The *Calendar* applet is a graphical calender tool supporting locally and remotely stored calendars. It supports multiple independent calendars in a single view, allowing to distinguish private from business appointments or to manage the activities of multiple family members.

As its backend and storage format, `remind(1)` is used. `remind(1)` is a mature and powerful command line tool supporting single events with and without times as well as many sorts of recurring events. The main advantage is the fact that a complete calendar is maintained in one *reminders* file with one event per line. The file is human-readable and can easily be edited by hand. Synchonization, merging and even revision control can easily be accomplished by standard tools such as `diff(1)`, `meld(1)` and `git(1)`. Several frontends exist for `remind` (`tkremind(1)`, for example), the *WallClock Calendar* is a new one with special support for multiple calendars.

The calendar files can be stored wherever the user likes – locally, on a home server or somewhere else. For any editing operation, *WallClock Calender* generates a `patch(1)` fragment and passes it to a user-configurable script, which is usually a `patch` command (see `calendar.cmdPatch`). The operation for reading a calendar file (usually defined as `cat <filename>`) is also user-configurable (see `calendar.cmdRead`).

This method for accessing calendar files allows a high degree of flexibility for where and how they are stored, and the use of `patch(1)` supports concurrent editing operations from multiple *WallClock* instances.

To use the *Calendar* applet, the `remind(1)` command line tool must be installed on the same computer as the *WallClock*.

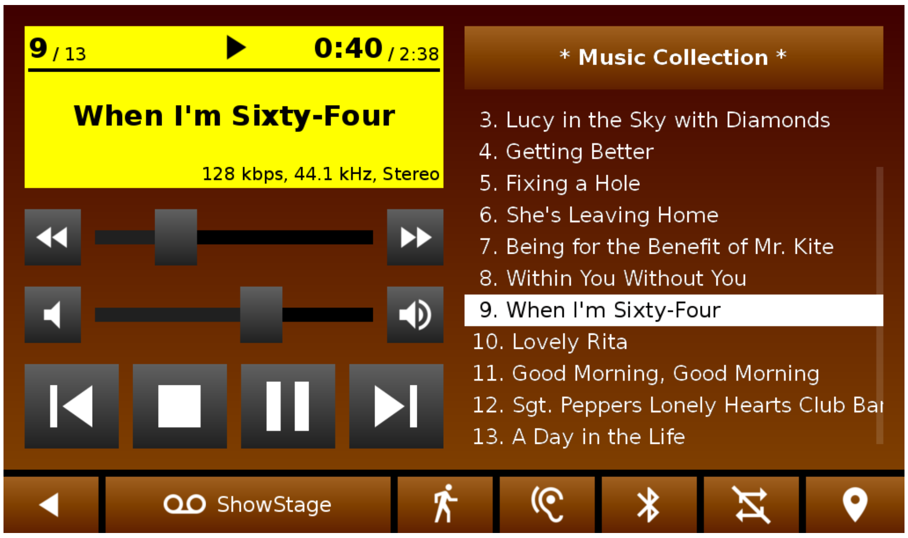
9.4 The Music Player

The *WallClock Music Player* is a front end for the *MPD* music player daemon (`www.musicpd.org`). It aims to support a home installations with multiple rooms, multiple users, and multiple virtual or physical stereo systems. In any room, any user shall be able to control any stereo system using any *WallClock* device. Everywhere in the house, he shall have access to the complete music collection and be able to get it transformed into acoustic air waves by any device he likes: hifi speakers in the living room, other speakers in the kitchen, earphones connected to the local device, or bluetooth speakers coupled with the *WallClock* device. Of course, a user can switch between them anytime and "take" the currently playing music with him as he moves to another room.

Technically, a "virtual stereo system" is an installed *MPD* instance running on some computer in the household. It has to be declared in `home2l.conf` using `music.<MPD>.host` and related parameters.

If the *MPDs* are configured with an http streaming output, the music can streamed back and played on the local device (*WallClock* must be compiled with *GStreamer* support for this).

Figure 9.2 shows a screenshot with the usual *MPD* controls on the left and a database and playlist browser on the right (the title bar can be pushed to navigate up or switch between the local collection and playlists).



3. Lucy in the Sky with Diamonds
4. Getting Better
5. Fixing a Hole
6. She's Leaving Home
7. Being for the Benefit of Mr. Kite
8. Within You Without You
9. When I'm Sixty-Four
10. Lovely Rita
11. Good Morning, Good Morning
12. Sgt. Peppers Lonely Hearts Club Ba
13. A Day in the Life

Figure 9.2: The *WallClock* Music Player

With the buttons in the bottom line you can (from left to right)

- select the "virtual stereo system" (*MPD* instance).
- take the currently playing music from one *MPD* instance to another (e.g. to seamlessly continue listening a song when moving from one room to another).
- select the output device (any output configured with *MPD* or stream to the local device, if available).
- enable or disable Bluetooth (Android).
- select the repeat mode.
- navigate to the currently playing song.

The `ui/mute` resource allows to mute the music player by automation rules, for example, if the doorbell rings.

A long push on the "Music" launcher on the main screen (Fig. 9.1) switches the music player on or off without switching to the music screen.

9.5 The Alarm Clock

The alarm clock can be enabled by pushing the clock display on the main screen. For each weekday, an individual alarm time can be programmed.

By default, the music player is activated on alarm. If the music player fails or the music is not loud enough (see `alarm.minLevelDb`), a ringing sound is played as a fallback.

For the case that the alarm clock fails completely, a dead-man script can be submitted to another computer, which can then wake you up with some other means (e.g. by a wakeup call to your mobile phone). See `alarm.extAlarmHost`, `alarm.extAlarmCmd` and `alarm.extAlarmDelay` for details.

9.6 List of Configuration Parameters

9.6.1 Parameters of Domain `ui`

`ui.passiveBehaviour` (bool) [= false] `wallclock/system.C:41`

Main application bahaviour.

Set the general application behaviour. If set to false (default), the app tries to control the display brightness, has own mechanisms for dimming the screen and tries to auto-activate after some time when the user shows no activity in another app (launcher-like behaviour).

If set to true, most of these mechanisms are disabled, and the app behaves like a normal app. All settings are controlled by the (Android) system.

`ui.standbyDelay` (int) [= 60000] `wallclock/system.C:52`

Time (ms) until standby mode is entered.

`ui.offDelay` (int) [= 3600000] `wallclock/system.C:55`

Time (ms) until the screen is switched off.

`ui.lightSensor.minLux` (float) [= 7] `wallclock/system.C:320`

Any Lux value below this will be rounded to this.

`ui.lightSensor.alOffset` (float) [= 20] `wallclock/system.C:323`

Linear part of the "apparent light" function (in Lux).

`ui.lightSensor.alFilterWeight` (float) [= 0.1] `wallclock/system.C:326`

Apparent light filter factor.

`ui.lightSensor.acThreshold` (float) [= 0.02] `wallclock/system.C:329`

Apparent change threshold to report a change.

The app tries to detect the presence of people in the room by monitoring the light sensor and waking up the app on a quick change in light. This and the previous parameters can be used to tune the sensitivity of this wakeup mechanism. Please refer to the source code to understand the exact algorithm.

Please do not expect too much – typical light sensors are not well suited for presence detection. If unsure, leave these settings with their defaults.

`ui.display.minLux` (float) [= 10.0] `wallclock/system.C:342`

Reference Lux value for the "minimum" brightness values.

`ui.display.typLux` (float) [= 100.0] `wallclock/system.C:345`

Reference Lux value for the "typical" brightness values.

`ui.display.maxLux` **(float)** [= 1000.0] `wallclock/system.C:348`

Reference Lux value for the "maximum" brightness values.

THe display brightness is adjusted according to a two-piece piece-wise linear function depending on the logarithm of the Lux value. This and the previous parameters define the Lux values for the three supporting points of the piece-wise linear function.

`ui.display.activeMin` **(float)** [= 0.5] `wallclock/system.C:358`

Minimum display brightness in active mode (percent).

`ui.display.activeTyp` **(float)** [= 0.7] `wallclock/system.C:361`

Typical display brightness in active mode (percent).

`ui.display.activeMax` **(float)** [= 1.0] `wallclock/system.C:364`

Maximum display brightness in active mode (percent).

`ui.display.standbyMin` **(float)** [= 0.25] `wallclock/system.C:368`

Minimum display brightness in standby mode (percent).

`ui.display.standbyTyp` **(float)** [= 0.35] `wallclock/system.C:371`

Typical display brightness in standby mode (percent).

`ui.display.standbyMax` **(float)** [= 0.5] `wallclock/system.C:374`

Maximum display brightness in standby mode (percent).

`ui.longPushTime` **(int)** [= 500] `wallclock/ui_base.C:51`

Time for a long push in milliseconds.

`ui.longPushTolerance` **(int)** [= 16] `wallclock/ui_base.C:54`

Tolerance for motion during a long push in pixels.

`ui.resizable` (bool) [= true] `wallclock/ui_base.C:57`

If set to 0, the UI window is not resizable.

`ui.audioDev` (string) `wallclock/ui_base.C:60`

Audio device for UI signaling (phone ringing, alarm clock).

`ui.daytimeRc` (string) [= "/local/timer/twilight/day"] `wallclock/app_home.C:56`

Resource (boolean) indicating daylight time.

`ui.accessPointRc` (string) [= "/local/accessPoint"] `wallclock/app_home.C:60`

Resource (boolean) for local (wifi) access point status display.

`ui.bluetoothRc` (string) [= "/local/ui/bluetooth"] `wallclock/app_home.C:63`

Resource (boolean) for local bluetooth status display.

`ui.weatherTempRc` (string) [= "/alias/weather/temp"] `wallclock/app_home.C:67`

Resource (temp) representing the outside temperature for the right info area (weather).

`ui.weatherData1Rc` (string) [= "/alias/ui/weatherData1"] `wallclock/app_home.C:70`

Resource for the upper data field of the right info area (weather).

`ui.weatherData2Rc` (string) [= "/alias/ui/weatherData2"] `wallclock/app_home.C:73`

Resource for the lower data field of the right info area (weather).

`ui.radarEyeRc` (string) [= "/alias/weather/radarEye"] `wallclock/app_home.C:77`

Resource for the radar eye as provided by the 'home2l-weather' driver.

`ui.lockSensor1Rc` (string) [= "/alias/ui/lockSensor1"] `wallclock/app_home.C:81`

Resource (bool) for the first lock display in the left info area (building).

`ui.lockSensor2Rc` (string) [= "/alias/ui/lockSensor2"] `wallclock/app_home.C:84`

Resource (bool) for the first lock display in the left info area (building).

`ui.motionDetectorRc` (string) [= "/alias/ui/motionDetector"] `wallclock/app_home.C:87`

Resource (bool) for the motion detector display in the left info area (building).

`ui.motionDetectorRetention` (int) [= 300000] `wallclock/app_home.C:90`

Retention time (ms) of the motion detector display.

`ui.radarEye.host` (string) `wallclock/app_home.C:97`

Host to run 'ui.radarEye.cmd' on. (OBSOLETE: Use string from URI "/alias/ui/radarEye" instead).

`ui.radarEye.cmd` (string) [= "cat $HOME2L_ROOT/tmp/weather/radarEye.pgm"] `wallclock/app_home.C:100`

Command to obtain .pgm file for the radar eye. (OBSOLETE: Use string from URI "/alias/ui/radarEye" instead).

`ui.launchMail` (string) `wallclock/app_home.C:254`

Android intent to launch a mail program (optional, Android only).

Only if set, a launch icon is shown on the home screen.

`ui.launchWeb` (string) `wallclock/app_home.C:259`

Android intent to launch a web browser (optional, Android only).

Only if set, a launch icon is shown on the home screen.

`ui.launchDesktop` (string) [= false] `wallclock/app_home.C:264`

If true, the home screen gets an icon to launch the Android desktop (Android only).

`ui.launchWeather` (string) `wallclock/app_home.C:267`

Android intent to launch a weather app (optional, Android only)..

If set, it will be launched if the weather area or radar eye are pushed.

9.6.2 Parameters of Domain `alarm`

`alarm.ringFile` (path) [= "share/sounds/alarm-classic.wav"] `wallclock/alarmclock.C:36`

Audio file to play if the music player fails or for pre-ringing.

`alarm.ringGap` (int) [= 0] `wallclock/alarmclock.C:40`

Number of milliseconds to wait before playing the ring file again.

`alarm.preRings` (int) [= 0] `wallclock/alarmclock.C:44`

Number of times the ring file is played before the music player is started.

`alarm.snoozeMinutes` (int) [= 10] `wallclock/alarmclock.C:48`

Number of snooze minutes.

`alarm.tryTime` (int) [= 15000] `wallclock/alarmclock.C:52`

Maximum time in milliseconds to try playing music.

If the music is not playing after this amount of time, the alarm clock reverts to ringing mode.

`alarm.minLevelDb` (int) [= -30] `wallclock/alarmclock.C:59`

Minimum level (DB) required for music.

If the music is below this level (e.g., because a radio station sends silence), the alarm clock reverts to ringing mode.

NOTE: This option only works if the music is output locally using GStreamer.

`alarm.extAlarmHost` (string) `wallclock/alarmclock.C:68`

Host to run an external alarm script on (local if unset).

This can be used to implement a fallback wakeup (e.g. by a wakeup phone call), if the wallclock fails for some reason.

`alarm.extAlarmCmd` (string) `wallclock/alarmclock.C:75`

Command to setup an external alarm.

This can be used to implement a fallback wakeup (e.g. by a wakeup phone call), if the wallclock fails for some reason. The command will be executed as follows:

```
<cmd> -i <hostname> <yyyy>-<mm>-<dd> <hh>:<mm>
```

`alarm.extAlarmDelay` (int) [= 3] `wallclock/alarmclock.C:84`

Delay of the external alarm setting.

Number of minutes n added to the set alarm time before transmitting the request to the external alarm resource.

In case of a failure in the "standby" or "snooze" state, the external alarm will go off n minutes after the time set. In case of a failure during alarming, the external alarm will go off between n and $2n$ minutes after the time set.

9.6.3 Parameters of Domain `var.alarm`

`var.alarm.enable` (bool) [= false] `wallclock/alarmclock.C:101`

Enable the alarm clock as a whole.

`var.alarm.timeSet.<n>` (int) `wallclock/alarmclock.C:105`

Wake up time set for week day $<n>$.

The time is given in minutes after midnight. Week days are numbered from 0 (Mon) to 6 (Sun).

Values < 0 denote that there is no alarm on the respective day. Values < -1 denote a hint to the UI if the alarm on that day is activated: The time is preset by the negated value.

`var.alarm.active` (int) [= 0] `wallclock/alarmclock.C:116`

Presently active alarm time (in minutes after the epoch)..

This variable is automatically set in a persistent way when an alarm goes off and set to 0 when the user switches off the alarm. It is used to recover the ringing state if the app crashes during alarm.

9.6.4 Parameters of Domain `phone`

`phone.enable` (bool) `wallclock/app_phone.C:62`

Enable the phone applet.

`phone.linphonerc` (path) `wallclock/app_phone.C:65`

Linphone RC file (Linphone backend only).

With the Linphone backend, all settings not directly accessible by Home2L settings are made in the (custom) Linphone RC file.

`phone.register` (string) `wallclock/app_phone.C:71`

Phone registration string.

`phone.secret` (string) `wallclock/app_phone.C:74`

Phone registration password.

`phone.ringFile` (path) [= "share/sounds/phone-classic.wav"] `wallclock/app_phone.C:77`

Ring tone file.

`phone.ringGap` (int) [= 2000] `wallclock/app_phone.C:80`

Number of milliseconds to wait between two rings..

`phone.rotation` (int) [= 0] `wallclock/app_phone.C:84`

Phone video rotation.

`phone.doorRegex` (string) `wallclock/app_phone.C:87`

Regex to decide whether a caller is a door phone.

`phone.ringFileDoor` (path) [= "share/sounds/dingdong-classic.wav"] `wallclock/app_phone.C:90`

Ring tone file for door phones calling.

`phone.openerDtmf` (string) `wallclock/app_phone.C:93`

DTMF sequence to send if the opener button is pushed.

`phone.openerRc` (string) `wallclock/app_phone.C:96`

Resource (type 'bool') to activat if the opener button is pushed.

`phone.openerDuration` (int) [= 1000] `wallclock/app_phone.C:99`

Duration of the opener signal.

`phone.openerHangup` (int) [= 0] `wallclock/app_phone.C:102`

Time until the phone hangs up after the opener button is pushed (0 = no auto-hangup).

`phone.fav<n>` (string) `wallclock/app_phone.C:110`

Define Phonebook entry $\#n$ ($n = 0..9$).

An entry has the form "[<display>|]<dial>", where <dial> is the number to be dialed, and (optionally) <display> is the printed name.

9.6.5 Parameters of Domain `calendar`

`calendar.enable` (bool) [= false] `wallclock/app_calendar.C:40`

Enable calendar applet.

`calendar.host` (string) `wallclock/app_calendar.C:43`

Host with calendar files (local if unset).

`calendar.dir` (string) `wallclock/app_calendar.C:46`

Storage directory for calendar (reminder) files..

`calendar.cmdRead` (string) [= "cat %s.rem"] `wallclock/app_calendar.C:49`

Command to read out a calendar.

The command may contain a "calendar name.

`calendar.cmdPatch` (string) [= "patch -ubNp1"] `wallclock/app_calendar.C:55`

Command to apply (patch) calendar changes.

The command will be executed on the host in the given dir, and the patch will be passed to its STDIN.

`calendar.<n>.name` (string) `wallclock/app_calendar.C:61`

Name for calendar $\#n$.

`calendar.<n>.color` (int) `wallclock/app_calendar.C:64`

Color for calendar $\#n$.

This should by given as a 6-digit hex number in the form `0x<rr><gg><bb>`.

9.6.6 Parameters of Domain `music`

`music.<MPD>.host` (string) `wallclock/app_music.C:52`

Network host name and optionally port of the given MPD instance..

This variable implicitly declares the server with its symbolic name <MPD>. If no port is given, the default port is assumed.

`music.port` (int) [= 6600] `wallclock/app_music.C:59`

Default port for MPD servers.

`music.<MPD>.password` (string) `wallclock/app_music.C:63`

Password of the MPD instance (optional, NOT IMPLEMENTED YET).

`music.(<MPD>|any)[.<OUTPUT>].name` (string) `wallclock/app_music.C:67`

Define a display name for an MPD server or an output.

`music.streamPort` (int) [= 8000] `wallclock/app_music.C:71`

Default port for HTTP streams coming from MPD servers.

The setting for a particular server <MPD> can be given by variable keyed "music.<MPD>.streamPort".

`music.streamBufferDuration` (int) [= 1000] `wallclock/app_music.C:78`

Buffer length [ms] for HTTP streaming..

`music.volumeGamma` (float) [= 1.0] `wallclock/app_music.C:82`

Gamma value for the volume controller (default and always used for local outputs).

The setting for a particular server <MPD> and optionally output <OUTPUT> can be given by variable keyed "music.<MPD>.[<OUTPUT>].volumeGamma".

`music.streamOutPrefix` (string) [= "stream"] `wallclock/app_music.C:89`

Name prefix for an output.

If the output name has the format "<prefix>[<port>]", it is recogized as a output for HTTP streaming, which can be listened to locally. For concenience, the port number can be appended to the stream prefix.

`music.recordOut` (string) [= "record"] `wallclock/app_music.C:97`

Name for a recording output.

If the output name has this name, it is recogized as an output with recording functionality. Such an output is not listed and selectable by the usual output functionality, but activated if and only if a streaming source is played.

`music.streamDirHint` (string) `wallclock/app_music.C:105`

MPD directory in which radio streams can probably be found.

The "go to current" button navigates to the parent directory of the currently playing song. If the song is not a local file, but a (HTTP) stream, this does not work out of the box. This setting optionally defines the directory to go to, if a non-file is currently played.

`music.recoveryInterval` (int) [= 2000] `wallclock/app_music.C:113`

Retry interval time if something (presently local streaming) fails..

`music.recoveryMaxTime` (int) [= 10000] `wallclock/app_music.C:117`

Maximum time to retry if something (presently local streaming) fails..

9.6.7 Parameters of Domain `var.music`

`var.music.server` (string) `wallclock/app_music.C:121`

MPD server to connect to first.

9.6.8 Parameters of Other Domains

`sync2l` (string) `wallclock/system.C:59`

Enable a Sync2l interface to the device's address book via a named pipe.

The argument defines the named pipe special file via which the device's address book can be accessed by the "sync2l" PIM synchronisation tool. If you do not know that tool (or do not use it), you should not set this parameter to avoid an unecessary security hole.

The pipe is created automatically, and it is made user and group readable and writable (mode 0660, ignoring an eventual umask). It is recommended to set the SGID bit of the parent directory and let it be owned be group 'home2l', so that a Debian or other chroot'ed Linux installation can access the pipe.

`sys.androidGateway` (string) `wallclock/home2l-wallclock.C:33`

Gateway IP adress for Android app.

If set, a default route is set with the given IP adress as a gateway by the Android app. This is useful if the device is conntected to the LAN in a way not directly supported by the Android UI, such as an OpenVPN tunnel over the USB cable.

9.7 List of Exported Resources

`ui/standby` (bool,wr) [= false] `wallclock/system.C:117`

Report and select standby mode.

If 'true', the screen is on, but eventually with reduced brightness (unless `ui/active` is also set). If neither `ui/active` nor `ui/standby` is 'true', the screen is switched off, and the device may enter a power saving mode, depending on the OS platform.

`ui/active` (bool,wr) [= false] `wallclock/system.C:126`

Report and select active mode.

If 'true', the screen is on at full brightness (as during interaction).

`ui/dispLight` (percent,ro) `wallclock/system.C:133`

Display brightness.

`ui/luxSensor` (float,ro) `wallclock/system.C:136`

Light sensor output in Lux.

`ui/mute` (bool,wr) [= false] `wallclock/system.C:140`

Audio muting.

If 'true', the music player is paused. This can be used to mute playing music if the doorbell rings or some other event in the house occurs which requires the attention of the user.

`ui/bluetooth` (bool,wr) `wallclock/system.C:149`

Report and set Bluetooth state.

`ui/bluetoothAudio` (bool,ro) `wallclock/system.C:152`

Report whether an audio device is connected via Bluetooth.

10 *DoorMan* – A Doorbell and Doorphone Service

10.1 Overview

DoorMan is a doorbell and doorphone service operated on the command line or as a background service. It must be linked with an IP phone library (presently *libLinphone*, *PJSIP* support is planned).

10.2 List of Configuration Parameters

`doorman.<ID>.enable` (bool) [= false] `doorman/home2l-doorman.C:29`

Define and enable a door phone with ID <ID>.

`doorman.<ID>.linphonerc` (string) `doorman/home2l-doorman.C:33`

Linphone RC file for door phone <ID> (Linphone backend only).

With the Linphone backend, all settings not directly accessible by Home2L settings are made in the (custom) Linphone RC file.

`doorman.<ID>.register` (string) `doorman/home2l-doorman.C:39`

Phone registration string.

`doorman.<ID>.secret` (string) `doorman/home2l-doorman.C:42`

Phone registration password.

`doorman.<ID>.rotation` (int) [= 0] `doorman/home2l-doorman.C:45`

Phone camera rotation.

`doorman.<ID>.buttonRc` (string) `doorman/home21-doorman.C:49`

External resource representing the bell button (optional; type must be 'bool').

There are two options to connect to a door button, which is either by defining an external resource using this parameter or by using the internal resource `doorman-ID/button`. If the external resource is defined, both resources are logically OR'ed internally.

`doorman.<ID>.buttonInertia` (int) [= 2000] `doorman/home21-doorman.C:57`

Minimum allowed time (ms) between two button pushes (default = 2000).

Button pushes are ignored if the previous push is less than this time ago.

`doorman.<ID>.dial` (string) `doorman/home21-doorman.C:63`

Default number to dial if the bell button is pushed.

`doorman.<ID>.openerRc` (string) `doorman/home21-doorman.C:67`

External resource to activate if the opener signal is received (optional).

`doorman.<ID>.openerDuration` (int) [= 1000] `doorman/home21-doorman.C:70`

Duration (ms) to activate the opener.

`doorman.<ID>.openerHangup` (int) [= 0] `doorman/home21-doorman.C:73`

Time (ms) after which we hangup after the opener was activated (0 = no hangup).

10.3 List of Exported Resources

`doorman-ID/button` (bool,wr) [= false] `doorman/home21-doorman.C:158`

Virtual bell button of the specified doorphone.

Driving this resource to true or false is equivalent to pushing or releasing a door bell button. To trigger a bell ring, a push and release event must occur. The "ID" in the driver name is replaced by the name of the declared phone.

There are two options to connect to a door button, which is either by defining an external resource using this parameter or by using the internal resource `doorman.<ID>.buttonRc`. Internally, both resources are logically OR'ed.

`doorman-ID/dial` (string,wr) `doorman/home21-doorman.C:172`

Number to dial for the specified doorphone.

This is the number dialed if the door button is pushed. The default value is set to the configuration parameter `doorman.<ID>.dial`. This resource allows to change the number to dial dynamically, for example, in order to temporarily redirect door bell calls to a mobile phone when out of home. The "ID" in the driver name is replaced by the name of the declared phone.

Index